Logic Functors: A Toolbox of Components for Building Customized and Embeddable Logics

Sébastien Ferré, Olivier Ridoux

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

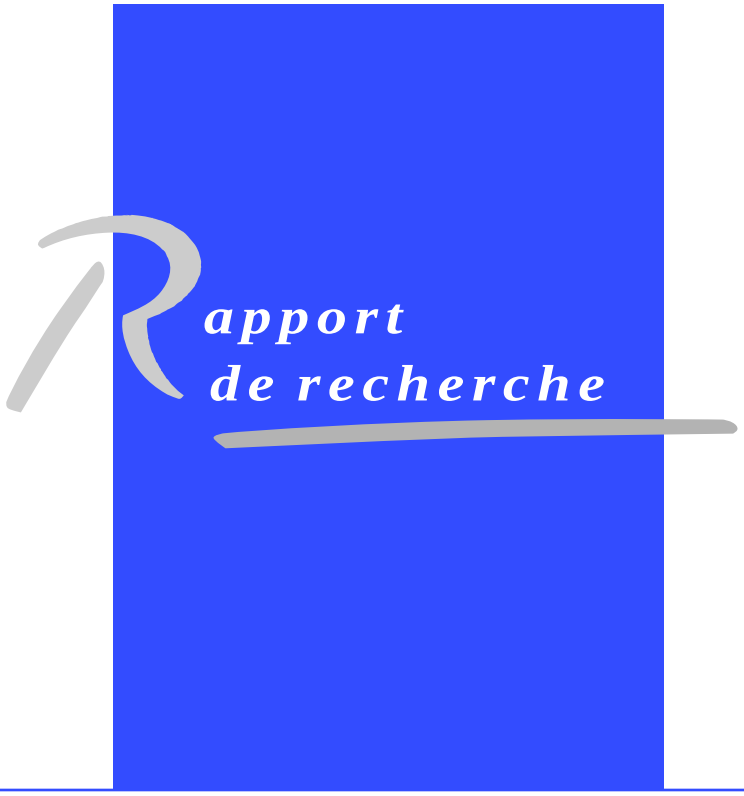
Logic Functors: A Toolbox of Components for Building Customized and Embeddable Logics

Sébastien Ferré and Olivier Ridoux

N°5871

Mars 2006

Systèmes symbolique



ISSN 0249-6399 ISRN INRIA/RR--5871--FR+ENG

Logic Functors: A Toolbox of Components for Building Customized and Embeddable Logics

Sébastien Ferré * and Olivier Ridoux †

Systèmes symbolique
Projet Lande

Rapport de recherche n ° 5871 — Mars 2006 — 103 pages

Abstract: Logic Functors form a framework for specifying new logics, and deriving automatically theorem provers and consistency/completeness diagnoses. Atomic functors are logics for manipulating symbols and concrete domains, while other functors are logic transformers that may add connectives or recursive structures, or may alter the semantics of a logic. The semantic structure of the framework is model theoretic as opposed to the verifunctional style often used in classical logic. This comes close to the semantics of description logics, and we show indeed that the logic $\mathcal{ALC}$ can be rebuilt using logic functors. This offers the immediate advantage that variants of $\mathcal{ALC}$ can be explored and implemented almost for free. This report comes with extensive appendices describing in detail a toolbox of logic functors (definitions, algorithms, theorems, and proofs).

Key-words: logic, components, modules and functors, theorem proves, type checking, application developpement

(Résumé : tsvp)

* ferre@irisa.fr
† ridoux@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Foncteurs logiques: Un jeu de composants pour construire des logiques spécialisées et embarquées

Résumé : Les foncteurs logiques forment un système permettant de spécifier de nouvelles logiques et d'en dériver automatiquement les démonstrateurs de théorèmes, ainsi que des diagnostiques de consistence/complétude. Les foncteurs atomiques sont des logiques manipulant des symboles et des domaines concrets, tandis que les autres foncteurs sont des transformateurs de logiques qui peuvent ajouter des connecteurs ou des structures récursives, ou encore modifier la sémantique d'une logique. Le style de la sémantique de ce système est celui de la théorie des modèles plutôt que le style vérifonctionnel souvent utilisée en logique classique. Ce style se rapproche de la sémantique des logiques de description, et nous montrons d'ailleurs que la logique $\mathcal{ALC}$ peut être reconstruite en utilisant les foncteurs logiques. Cela offre l'avantage immédiat que des variantes de $\mathcal{ALC}$ peuvent être explorées et implémentées presque gratuitement. Ce rapport est accompagné de nombreuses annexes décrivant en détail une boîte à outils de foncteurs logiques (définitions, algorithmes, théorèmes et preuves).

Mots-clé : logique, composants, modules et foncteurs, démonstration de théorèmes, vérification de type, développement d'applications

Chapter 1

Introduction

We present a framework for building embeddable automatic theorem provers for *customized logics*. The framework defines *logic functors* as logic components; for instance, one component may be the propositional logic, another component may be the interval logic, also called intervals. Logic functors can be composed to form new logics, for instance, propositional logic on intervals.

Each logic functor has its own proof-theory, which can be implemented as a theorem prover. We desire that the proof-theory and the theorem prover of the composition of logic functors should result from the composition of the proof-theories and the theorem provers of the component logic functors.

All logic functors and their compositions implement a common *interface*. This makes it possible to construct generic applications that can be instantiated with a logic component. Conversely, customized logics built using the logic functors can be *embedded* in an application that comply with this interface.

Logic functors specify software components off-the-shelf (COTS), the validation of the composition of which reduces to a form of type-cheking, and their composition automatically results in an automatic theorem prover. Logic functors can be assembled by laymen, and used routinely in system-level programming, such as compilers, operating systems, file-systems, and information systems.

An implementation of all logic functors in the appendices (and more) is available as an Open Source software library, available at `http://www.irisa.fr/lande/ferre/logfun`. Properties of built logics can be checked automatically. Logic functors are implemented as OCaml module functors.

1.1 Logic-based information processing systems

In [FR04, Fer02], we have proposed Logical Information Systems that are built upon a variant of Formal Concept Analysis [GW99], namely Logical Concept Analysis [FR00]. The

framework is generic in the sense that any logic whose deduction relation forms a lattice can be plugged-in. However, if one leaves the logic totally undefined, then one puts too much responsibility on the end-users or on a knowledge-base administrator. It is unlikely they can design such a logic component themselves. By using the framework developed in this article, one can design a toolbox of logic components, and the user has only the responsibility of composing those components. The design of these Logical Information Systems is the main motivation for this research.

However, we believe the application scope of this research goes beyond our Logical Information Systems. Several information processing domains have logic-based components in which logic plays a crucial role: e.g., logic-based information retrieval [SM83, vRCL98], logic-based diagnosis [Poo88], logic-based programming [Llo87, MS98], logic-based program analysis [SFRW98, AMSS98, CSS99]. These components model an information processing domain in logic, and also they bring to the front solutions in which logic is the main engine. This can be illustrated by the difference between using a logic of programs and programming in logic.

Even in information processing domains where traditionally logic does not play a crucial role it has been proposed to embed a logic component in otherwise not logic-based systems. For instance, in [IB96] the authors propose to model quality of service (QoS) conditions in logic, and to make applications check dynamically that the platform on which they run enforces the condition for a specified quality of service.

The logic in use in these system is often not defined by a single pure deduction system, but rather it combines several logics together. The designer of an application has to make an *ad hoc* proof of consistency and an *ad hoc* implementation (i.e., a theorem prover) every time he designs a new *ad hoc* logic. Since these logics are often variants of a more standard logic we call them *customized logics*.

In order to favour separation of concerns, it is important that the application that is based on a logic engine, and the logic engine itself, be designed separately. This implies that the interface of the logic engine should not depend on the logic itself. This is what we call *embeddability* of the logic component.

For instance, practical query-answering systems often use a mixture of logic and concrete computations. Queries are built with logical connectives, and with purely operational constructs like wild-cards. This usually causes no harm because the query-answering system is not actually a theorem prover, and thus does not actually implement a logic. Indeed, in most cases a boolean query is used to filter concrete strings that contain no wild-cards, and no boolean connectives.

However, one can imagine a logic-based query-answering systems in which queries and data actually use the same language. For instance, one may use the full language with connectives and wild-cards both for describing entries in an information system, and for querying them [FR04]. Then the query-answering system must decide something which can be written as $description \sqsubseteq query$ (where $\sqsubseteq$ means logical consequence of the logic used in descriptions and queries), and it must have the full capacity of a theorem prover for a logic whose syntax is the description/querying language. The choice of a particular logic depends

on the application, but query-answering is generic and depends solely on the existence of $\sqsubseteq$.

If we need to separately design the application and its logic components, then who should develop the embedded logic components ?

1.2 The actors of the development of an information processing system

In this section, we present our views on the Actors of the development on an information processing system. Note that Actors are not necessarily incarnated in one person; each Actor may gather several persons possibly not living at the same time. In short, Actors are roles, rather than persons. Sometimes, Actors may even be incarnated in computer programs.

What follows is rather standard in the information system (especially data-base) community because it has adopted organisation standards of industry and administration, but we think it is not widely accepted by the academic community for other kinds of information processing systems, where it tends to follow more academic standards in which several Actors are collapsed into one, the Researcher.

The first Actor is the Theorist; he invents an abstract framework, like, for instance, relational algebra, lattice theory, or logic.

If the abstract framework has applications, then a second Actor, the System Programmer, implements (part of) the theory in a *generic* system for these applications. This results in systems like data-bases, static analysers, or logic programming systems.

Then the third Actor, the Application Designer, applies the abstract framework to a concrete objective by *instantiating* a generic system. This can be done by composing a data-base schema, or a program property, or a logic program.

Finally, the User, the fourth Actor, maintains and uses an application. He queries a data-base, he analyses programs, or he runs logic programs. The User is often incarnated in programs.

Certainly, the User could be analysed further in Administrators, End-Users, etc. However, we stop here because it is the relation between the System Programmer and the Application Designer that interests us; the first one creates a generic system, and the second one instantiates it.

1.3 Genericity and instantiation

Genericity is often achieved by designing a language: e.g., a data-base schema language, a lattice operation language, and a programming language. Correspondingly, instantiation is done by programming and composing: e.g., drawing a data-base schema, composing an abstract domain for static analysis, or composing a logic program.

We propose to do the same for logic-based tools. Indeed, on one hand the System Programmer is competent for building a logic subsystem, but he does not know the application;

he only knows the range of applications. On the other hand the Application Designer knows the application, but is generally not competent for building a logic subsystem. In this article, we will act as System Programmers by providing elementary components for safely building a logic subsystem, and also as Theorists by giving formal results on the composition laws of these components.

We explore how to systematically build logics using basic components that we call *logic functors*. By "construction of a logic" we mean the definition of its syntax, its semantics, and its abstract implementation as a deduction system. All logic functors we describe in this article have also a concrete implementation as an actual module. As this implementation is based on module functors of OCaml, logic functors can be directly composed in OCaml itself, so there is no need to program a *composer*. In fact the language for composing functors is simple, and there is no need to learn the OCaml language in order to build customized logics. Examples are given in Section 4

1.4 Customized logics

The range of logic functors can be very large. In this article we consider for instance products and sums of logics, propositions (on arbitrary formulas), intervals, multisets, some concrete domains like integers or strings (e.g., "begin with", "contains"), *AIK* (a modal epistemic logic [Lev90]). See the appendices for a full list.

The whole framework is geared towards manipulating logics as partial orderings, where the ordering is a specialization/generalization relation. This relation receives various names, depending on the context: deduction, entailment, subsumption. The latter, which is found in description logics, has our preference as it betters suggests the specialization/generalization ordering. This requirement for partial orderings excludes non-monotonic logics. Note that non-monotonicity is seldom a goal in itself, and that notoriously non-monotonic features have a monotonic rendering; e.g., Closed World Assumption can be reflected in the monotonic modal logic *AIK*.

We will consider as a motivating example an application for dealing with bibliographic entries. Each entry has a description made of its author name(s), title, type of cover, publisher, and date. The User navigates through a set of entries by comparing descriptions with queries that are written in the same language. The application answers navigation queries by lists of entries that match the queries, and lists of subqueries that can complement the current one to form a more precise query. So doing, we have a logic-based notion of navigation where matching a query is being in some place, and subqueries are links to other places. For instance, let us assume the following entry set:

- `descr(entry`$_1$`) =`
  `author: "Kipling", title: "The Jungle Book", paper-back, publisher:`
  `"Penguin", year: 1985,`

- `descr(entry`$_2$`) =`
  `author: "Kipling", title: "The Jungle Book", hard-cover, publisher: "Century Co.", year: 1908,`
- `descr(entry`$_3$`) =`
  `author: "Kipling", title: "Just So Stories", hard-cover, year: 1902.`

An answer to the query:
`title: contains "Jungle"`
is:

`hard-cover`	`publisher:`	`"Century Co."`	`year:`	`1900..1950`
`paper-back`	`publisher:`	`"Penguin"`	`year:`	`1950..2000`

because several entries (`entry`$_1$ and `entry`$_2$) have a description that entails the query (i.e., they are possible answers), and the application asks the user to make his query more specific by suggesting some relevant refinements. Note that `author is "Kipling"` is not a relevant refinement because it is true of all matching entries. For every possible answer `entry` we have `descr(entry)`$\sqsubseteq$`query`, and for every relevant refinement `x` the following holds

1. there exists a possible answer `e`$_1$ such that `descr(e`$_1$`)`$\sqsubseteq$`x`, and
2. there exists a possible answer `e`$_2$ such that `descr(e`$_2$`)`$\not\sqsubseteq$`x`.

In other words, a refinement must restrict the set of possible answers while avoiding to make it empty. We will not go any further in the description of this application (see [FR04]). We simply note that:

1. descriptions, queries, and answers belong to the same logical language, which combines logical symbols and expressions such as strings, numbers, or intervals, and
2. one can design a similar application with a different logic, e.g., for manipulating software components. Thus, it is important that all different logics share a common interface for being able to separately write programs for the navigation system and for the logic subsystem it uses.

1.5 Summary

We define tools for building *automatic* theorem provers for *customized logics* for allowing Users who are not sophisticated logic actors. Note also that the User may be a program itself: e.g., a mobile agent running on a host system [IB96]. This rules out interactive theorem provers.

Validating a theorem prover built by using our tools must be as simple as possible. Some kind of type-checking is ideal because the Application Designer, though it may be more sophisticated than the User, is not a logic actor.

Finally, the resulting theorem provers must have a common interface so that they can be *embedded* in generic applications. Subsumption algorithms terminate in all the logic components that we define. Thus, the logic components can be safely embedded in applications as black-boxes.

This article is organized as follows. Chapter 2 introduces the notions of logics and logic functors, which are based on definitions for syntax, semantics, operations, and logic properties. Chapter 3 details the concrete implementation of logic functors as ML functors. Chapter 4 examplifies the use of logic functors, and Chapter 5 presents related work, concludes and states future work. The numerous appendices give the full and detailed description of logic functors and combinators. They constitute a first toolbox of customized and embeddable logic components. A large part of these appendices is made of proofs of logic properties.

Chapter 2

Logics and Logic Functors

If an Application Designer has to define a customized logic by the means of composing primitive components, these components should be of a "high-level", so that the resulting logic subsystem can be proven to be correct. Indeed, if the primitive components are too low-level, proving the correctness of the result is similar to proving the correctness of a program. Thus, we decided to define logical components that are very close to be logics themselves. So doing, we abandon computational expressivity, but we will see that proving correctness is no more than type-checking.

Our idea is to consider that a logic interprets its formulas as functions of their atoms. By abstracting atomic formulas from the language of a logic we obtain what we call a *logic functor*. A logic functor can be applied to a logic to generate a new logic. For instance, if propositional logic is abstracted over its atomic formulas, we obtain a logic functor called *Prop*, which we can apply to, say, a logic on integer intervals, $Interval(Int)$, to form propositional logic on intervals, $Prop(Interval(Int))$.

In Section 2.1 we first define the syntax and the semantics of a logic, which is a classical way of specifying a logic. The (abstract) syntax tells what kind of formulas can be expressed and manipulated, and the semantics tells what meaning can be given to these formulas. Then in Section 2.2 we introduce a set of operations that apply on formulas. These operations are defined as functions working only at the syntactical level, which makes them directly implementable, and constitute the executable version of the logic. In order for these operations to make sense w.r.t. the semantics, properties relating the behaviour of operations with semantics are associated to these operations. The statement of properties on logics requires proofs, combinations of properties play the role of types, and theorems on properties play the role type-checking rules. Finally, in Section 2.3.1, logics are defined as the combination of syntax, semantics, and operations; and logic functors are defined as functions from logics to logics.

2.1 Syntax and Semantics

Syntax is often defined from a signature of atoms and connectors. However, in order to allow for concrete domains and concrete structures, we choose to have a more general definition of syntax. Here we do not bother about the concrete syntax as it is just a matter of pretty-printing and parsing compared to the abstract syntax which consider formulas as set-theoretical structures (e.g., pairs, finite sets, intervals).

Definition 1 (Syntax) *An* abstract syntax AS *is an enumerable set of structures, the element of which are called* formulas*.*

As our logics are mainly designed for information systems, it is useful to distinguish two subsets of syntax:

- $TELL \subseteq AS$ is the subset of formulas that can be used in object descriptions,
- $ASK \subseteq AS$ is the subset of formulas that can be used in queries.

No constraint is put on these two subsets, except they should not be empty: e.g., they can be disjoint or not, they can be equal to AS or not. Wen they are not defined explicitly, they are equal to the full set of fomulas AS. The distinction between $TELL$-formulas, and ASK-formulas is important w.r.t. logic properties, as shown in Section 2.2, because requirements from information systems may not be the same on them.

Now, it is important to attach a semantics to a logic in order to give a meaning to formulas and operations. Like in description logics, formulas are not interpreted by truth-values, but by sets of interpretations (or individuals). Formulas are given a semantics by linking them to *interpretations*. These interpretations can be understood as possible objects, and reciprocally, formulas can be understood as expressing a constraint/filter/pattern on possible objects. Moreover, it is possible to define a partial ordering on interpretations, which is useful for instance to define intervals in a generic way.

Definition 2 (Semantics) *Given a syntax* AS*, a* semantics S *based on* AS *is a pair* $(I, \models, \leq)$*, where*

- I *is a set of* interpretations*, the* interpretation domain*,*
- $\models \subseteq I \times AS$ *is a* satisfaction relation *between interpretations and formulas,*
- $\leq$ *is a partial ordering on the set* I*.*

$i \models f$ *reads "*i *is a* model *of* f*". For every formula* $f \in AS$*,* $M(f) = \{i \in I \mid i \models f\}$ *denotes the set of all models of formula* f*. For every formulas* $f, g \in AS$*, an* entailment relation *is defined as "*f *entails* g*" iff* $M(f) \subseteq M(g)$*.*

The entailment relation is never used formally in this report, but we believe it provides a good intuition for the frequent usage in proofs of the inclusion of sets of models.

2.2 Operations and Properties

Operations define the interface through which a logic can be used by programs. Properties are about the adequation of these operations w.r.t. semantics.

The formulas define the language of the logic, the semantics defines its interpretation, and an *implementation* defines how the logic implements an *interface* that is common to all logics. This common interface is made of a set of operations, which are defined below. The most important operation is the *subsumption*. It tests whether a formula is more specific than another. Because of the similarities between our framework and description logics, we reuse their names and notations for operations (e.g., $\sqsubseteq$ for subsumption).

There is no constraint, except for their types, on what operations can do. So, we define for each operation properties that relate the semantics and the operations of a logic.

For each operation, we explain its purpose, its type, its default definition, and we define a set of properties that apply to it. Properties may receive several definitions corresponding to different scopes: one point of the operation domain, the entire operation domain, and intermediate domains based on *TELL*, and *ASK*. The properties of default definition are given beside them.

2.2.1 Top $\top$

The purpose of the *top* is to return the most general formula of the logic. It usually is a *ASK*-formula, and not a *TELL*-formula. It may be undefined. When defined, its expected meaning is that it has every interpretation as a model.

type: $AS \cup \{undef\}$.

defined:

$$def_{\top} =_{def} \top \neq undef$$

complete:

$$cp_{\top} =_{def} def_{\top} \Rightarrow M(\top) = I$$

default: $\top_d =_{def} undef$

$cp_{\top_d}$

2.2.2 Bottom $\bot$

The purpose of the *bottom* is to return the least general formula of the logic. It usually is neither a *TELL*-formula, nor a *ASK*-formula. So, when defined, its role is mainly to make possible the proof of some properties. Its expected meaning is that it has no model.

type: $AS \cup \{undef\}$.

defined:

$$def_{\bot} =_{def} \bot \neq undef$$

consistent:
$$cs_{\bot} =_{def} def_{\bot} \Rightarrow M(\bot) = \emptyset$$

default: $\bot_d =_{def} undef$
$$cs_{\bot_d}$$

2.2.3 Subsumption $\sqsubseteq$

The purpose of *subsumption* is to test whether a formula is more specific than another. This is the most useful operation as it is used in information systems to compute answers to a query q. An object o is such an answer if its description $d(o)$ is subsumed by the query q ($d(o) \sqsubseteq q$). Its expected meaning is to be equivalent to the inclusion of sets of models.

type: $AS \times AS \rightarrow bool$.

consistent in $(f : AS, g : AS)$**:**
$$cs_{\sqsubseteq}(f, g) =_{def} f \sqsubseteq g \Rightarrow M(f) \subseteq M(g)$$
$$cs_{\sqsubseteq} =_{def} \forall f, g \in AS : cs_{\sqsubseteq}(f, g)$$

complete in $(f : AS, g : AS)$**:**
$$cp_{\sqsubseteq}(f, g) =_{def} M(f) \subseteq M(g) \Rightarrow f \sqsubseteq g$$
$$cp_{\sqsubseteq} =_{def} \forall f, g \in AS : cp_{\sqsubseteq}(f, g)$$
$$cp'_{\sqsubseteq} =_{def} \forall d \in TELL, x \in ASK : cp_{\sqsubseteq}(d, x)$$

default: $f \sqsubseteq_d g =_{def} false$
$$cs_{\sqsubseteq_d}$$

In order to avoid false positives (an answer that should not be an answer), information systems usually require consistency $cs_{\sqsubseteq}$. To avoid false negatives (missed answers) it is sufficient to have the partial completeness $cp'_{\sqsubseteq}$, which is true is more logics than full completeness, and allows for more efficient implementations. This is where the motivation for distinguishing $TELL$-formulas, and ASK-formulas.

2.2.4 Conjunction $\sqcap$

The purpose of *conjunction* is to sum up the information of 2 formulas in one, i.e. to return the most general formula that is subsumed by the two argument formulas. It can be partially defined. When the two argument formulas are incompatible, it should return the contradiction, if defined. Its expected meaning is to be equivalent to the intersection of sets of models.

type: $AS \times AS \rightarrow AS \cup \{undef\}$.

defined in $(f : AS, g : AS)$**:**
$$def_{\sqcap}(f, g) =_{def} f \sqcap g \neq undef$$
$$def_{\sqcap} =_{def} \forall f, g \in AS : def_{\sqcap}(f, g)$$
$$defst_{\sqcap} =_{def} \forall f, g \in AS : M(f) \cap M(g) \neq \emptyset \Rightarrow def_{\sqcap}(f, g)$$

consistent in $(f : AS, g : AS)$**:**
$cs_{\sqcap}(f,g) =_{def} def_{\sqcap}(f,g) \Rightarrow M(f \sqcap g) \subseteq M(f) \cap M(g)$
$cs_{\sqcap} =_{def} \forall f, g \in AS : cs_{\sqcap}(f,g)$

complete in $(f : AS, g : AS)$**:**
$cp_{\sqcap}(f,g) =_{def} def_{\sqcap}(f,g) \Rightarrow M(f \sqcap g) \supseteq M(f) \cap M(g)$
$cp_{\sqcap} =_{def} \forall f, g \in AS : cp_{\sqcap}(f,g)$

default: $f \sqcap_d g =_{def} undef$
$cs_{\sqcap_d} \wedge cp_{\sqcap_d}$

When the conjunction is totally undefined (default behaviour), it is both consistent and complete. The need for making conjunction more defined usually comes from property proofs, especially about reducedness (Section 2.2.8). Defining conjunction can also be useful for compacting queries or other combinations of formulas.

2.2.5 Disjunction $\sqcup$

The purpose of *disjunction* is to generate the most general formulas whose models are all models of either argument formulas. Its expected meaning is to be equivalent to the union of models, as shown by consistency and completeness properties.

type: $AS \times AS \rightarrow \mathcal{P}(AS)$.

consistent in $(f : AS, g : AS)$**:**
$cs_{\sqcup}(f,g) =_{def} \bigcup_{h \in f \sqcup g} M(h) \subseteq M(f) \cup M(g)$
$cs_{\sqcup} =_{def} \forall f, g \in AS : cs_{\sqcup}(f,g)$

complete in $(f : AS, g : AS)$**:**
$cp_{\sqcup}(f,g) =_{def} \bigcup_{h \in f \sqcup g} M(h) \supseteq M(f) \cup M(g)$
$cp_{\sqcup} =_{def} \forall f, g \in AS : cp_{\sqcup}(f,g)$

default: $f \sqcup_d g =_{def} \{f, g\}$
$cs_{\sqcup_d} \wedge cp_{\sqcup_d}$

2.2.6 Lower ordering $\leqslant$

The purpose of *lower ordering* is to test whether a formula starts before another formula, according to the partial ordering on interpretations. The simplest example is probably intervals on integers, where interpretations are integers, and the partial ordering is the natural one. The interval $[1, 5]$ starts before the interval $[3, 4]$ because the lowest model of the latter is smaller than the lowest model of the former.

type: $AS \times AS \rightarrow bool$.

consistent in $(f : AS, g : AS)$**:**
$cs_{\leqslant}(f,g) =_{def} f \leqslant g \Rightarrow \forall j \in M(g) : \exists i \in M(f) : i \leq j$
$cs_{\leqslant} =_{def} \forall f, g \in AS : cs_{\leqslant}(f,g)$

complete in $(f : AS, g : AS)$**:**
$cp_{\leqslant}(f,g) =_{def} (\forall j \in M(g) : \exists i \in M(f) : i \leq j) \Rightarrow f \leqslant g$
$cp_{\leqslant} =_{def} \forall f, g \in AS : cp_{\leqslant}(f,g)$

default: $f \leqslant_d g =_{def} false$
$cs_{\leqslant_d}$

2.2.7 Upper ordering $\leqslant$

The purpose of the *upper ordering* is to test whether a formula ends before another formula, according to the partial ordering on interpretations. In the above example, the interval $[3,4]$ ends before the interval $[1,5]$.

type: $AS \times AS \rightarrow bool$.

$\leqslant$ **is consistent in** $(f : AS, g : AS)$**:**
$cs_{\leqslant}(f,g) =_{def} f \leqslant g \Rightarrow \forall i \in M(f) : \exists j \in M(g) : i \leq j$
$cs_{\leqslant} =_{def} \forall f, g \in AS : cs_{\leqslant}(f,g)$

$\leqslant$ **is complete in** $(f : AS, g : AS)$**:**
$cp_{\leqslant}(f,g) =_{def} (\forall i \in M(f) : \exists j \in M(g) : i \leq j) \Rightarrow f \leqslant g$
$cp_{\leqslant} =_{def} \forall f, g \in AS : cp_{\leqslant}(f,g)$

default: $f \leqslant_d g =_{def} false$
$cs_{\leqslant_d}$

2.2.8 Syntax vs. Semantics

Some properties are not related to one operation, but to the syntax or to the set of all operations. Before defining a list of such properties, we first need to extend the definitions of syntax and subsumption to sets of formulas.

Definition 3 *Given $\mathcal{P}(X)$ denotes the powerset of X:*

- $AS^* =_{def} \mathcal{P}(AS)$,
- $ASK^* =_{def} \mathcal{P}(ASK)$,
- $TELL^* =_{def} \{F \in \mathcal{P}(\{\bot\} \cup TELL \cup ASK) \mid F \cap \{\bot\} \cup TELL \neq \emptyset\}$,
- $F \sqsubseteq^* G =_{def} \bigvee \left\{ \begin{array}{l} \exists f \in F, g \in G : f \sqsubseteq g \\ \exists f \in F : f \sqsubseteq \bot \\ \exists g \in G : \top \sqsubseteq g \end{array} \right.$

- $closed_{\sqcap}(F) =_{def} \forall f \neq f' \in F : f \sqcap f' = undef$,
- $closed_{\sqcup}(G) =_{def} \forall g \neq g' \in G : g \sqcup g' \subseteq G$.

all descriptors are features:
$df =_{def} TELL \subseteq ASK$

satisfiable in $f : AS$: f has a model.
$st(f) =_{def} M(f) \neq \emptyset$
$st =_{def} \forall f \in AS : st(f)$
$st' =_{def} \forall d \in TELL : st(d)$

singleton in $f : AS$: f has one and only one model.
$sg(f) =_{def} |M(f)| = 1$
$sg =_{def} \forall f \in AS : sg(f)$
$sg' =_{def} \forall d \in ASK : sg(d)$

reduced in $(F : AS^*, G : AS^*)$:
$reduced(F, G) =_{def} closed_{\sqcap}(F) \wedge closed_{\sqcup}(G) \Rightarrow$
$(\bigcap_{f \in F} M(f) \subseteq \bigcup_{g \in G} M(g) \Rightarrow F \sqsubseteq^* G)$
$reduced =_{def} \forall F, G \in AS^* : reduced(F, G)$
$reduced' =_{def} \forall F \in TELL^*, G \in ASK^* : reduced(F, G)$
$reduced.left =_{def} \forall F \in AS^*, g \in AS : reduced(F, \{g\})$
$reduced.right =_{def} \forall f \in AS, G \in AS^* : reduced(\{f\}, G)$
$reduced.top =_{def} \forall g \in AS : reduced(\emptyset, \{g\})$
$reduced.bot =_{def} \forall f \in AS : reduced(\{f\}, \emptyset)$
$reduced.bot' =_{def} \forall d \in TELL : reduced(\{d\}, \emptyset)$

Note that *reduced.left* entails *reduced.top*, and that *reduced.right* entails *reduced.bot*, which entails *reduced.bot'*. Note also that $reduced(\emptyset, \emptyset)$ is always true (provided the interpretation domain is not empty).

2.2.9 Comments

Properties are rather technical, because they have been introduced progressively in order to prove some properties on logics and logic functors. This assumes we have a starting point, i.e. some initial properties we are interested in. Indeed, in the context of information systems, we need to answer queries. In Section 2.2.3 we already show that consistency $cs_{\sqsubseteq}$ and partial completeness ${cp'}_{\sqsubseteq}$ are required on subsumption in order to answer queries correctly.

As our logics are built by composing sub-logics by logic functors, it is important to understand that the properties required on each sub-logics will not be the same, and will depend on each logic functor. For instance, a logic functor may enforce a property even if it is not satisfied by sub-logics. The other way round, some properties may be required on sub-logics, even if it is not so in the built logic. This explains why we introduce much more properties than needed on the logics used in information systems.

Operations $\sqcap$, $\sqcup$, $\top$, $\bot$ are all defined on the syntax of some logic, but they are not necessarily connectives of the logic, simply because the connectives of a logic are part of the syntax, and may be different from these operations. Similarly, the syntax and the semantics may define negation or quantifiers, though they are absent from the set of operations.

Note that some operations (conjunction, tautology, and contradiction) can be only partially defined (by using *undef*) if it is convenient, and that their properties apply only on the domain where they are defined. So it is easy to make them satisfy these properties, like consistency and completeness, by keeping them undefined. Reducedness counterbalances this triviality by requiring these operations to be defined *enough*. And this property is required in some logic functors in order to achieve subsumption completeness, which is crucial as said above.

2.3 Logics and Logic Functors

In this section we formally define the class of logics, and then the class of logic functors, which are functions from logics to logics.

2.3.1 Logics

A *logic* is the association of an abstract syntax, a semantics, an implementation, and a type made of a set of properties. The class of all logics is denoted by $\mathbb{L}$.

Definition 4 (Logic) *A logic L is a tuple (AS_L, S_L, P_L, T_L), where AS_L is (the abstract syntax of) a set of formulas, S_L is a semantics based on AS_L, P_L is an implementation, i.e. a set of operations, based on AS_L, and T_L is the* type *of the logic, i.e., a set of properties that are satisfied by operations w.r.t. semantics.*

When necessary, the satisfaction relation $\models$ of a logic L will be written $\models_L$, the interpretation domain I will be written I_L, the models $M(f)$ will be written $M_L(f)$, and each operation op will be written op_L.

2.3.2 Logic functors

We have just defined logics. *Logic functors* take logics as parameters and return a logic. Logic functors also have a syntax, a semantics, a set of operations, and a type, but they are all abstracted over one or more logics that are considered as formal parameters. For instance, the logic functor $Prop(X)$ is the propositional logic, whose atoms have been abstracted by the formal parameter X, and can thus be replaced by more complex formulas. Then the classic propositional logic is reconstructed by applying the Composer to the expression $Prop(Atom)$; and the composed logic $Prop(Interval(Int))$ replaces the classic atoms by intervals on integers. This process is very similar to the composition of mathematic expressions from operations (e.g., $+, -, \times$), or complex programming types from basic types (e.g., $int, bool$) and type constructors (e.g., $array, list$). It is also possible to define a new

logic functor as a parameterized composition of existing functors. We call such a functor a *combinator*: e.g., $\lambda X.Prop(Interval(X))$.

We formally define the class of logic functors. Given $\mathbb{L}$ the class of logics, the class of logic functors $\mathbb{F}$ includes all functions from tuples of logics to logics. The special case of logic functors with arity 0 corresponds to the class of logics $\mathbb{L}$. The notation F/n is used to say that F is a logic functor with arity n, i.e., expecting n logics in order to produce a logic.

Let $\mathbb{AS}$ be the class of all syntaxes, $\mathbb{S}$ be the class of all semantics, $\mathbb{P}$ be the class of all sets of operations, and $\mathbb{T}$ the class of all logic types. The syntax of a logic functor is simply a function from the syntaxes of the logic functors which are its arguments, to the syntax of the resulting logic.

Definition 5 (Logic Functor) *A* logic functor F *is a tuple* (AS_F, S_F, P_F, T_F) *where*

- *the abstract syntax* AS_F *is a function of type* $\mathbb{AS}^n \to \mathbb{AS}$, *such that* $AS_{F(L_1,\ldots,L_n)} = AS_F(AS_{L_1}, \ldots, AS_{L_n})$;
- *the semantics* S_F *is a function of type* $\mathbb{S}^n \to \mathbb{S}$, *such that* $S_{F(L_1,\ldots,L_n)} = S_F(S_{L_1}, \ldots, S_{L_n})$;
- *the implementation* P_F *is a function of type* $\mathbb{P}^n \to \mathbb{P}$, *such that* $P_{F(L_1,\ldots,L_n)} = P_F(P_{L_1}, \ldots, P_{L_n})$;
- *the type* T_F *is a function of type* $\mathbb{T}^n \to \mathbb{T}$, *such that* $T_{F(L_1,\ldots,L_n)} = T_F(T_{L_1}, \ldots, T_{L_n})$.

In order to illustrate this abstract notion of logic functor, and prepare examples given in next sections, we shortly describe a few logic functors (the detailed description of our logic functors are available in the appendices). We classify them according to their role in the composition of logics:

Initiators are 0-ary functors representing various concrete domains

- *Unit*/0: the logic having a single formula, and a single interpretation,
- *Atom*/0: the classical atoms,
- *Int*/0: integers,
- *Card*/0: the decreasing chain of naturals for expressing "at least n" patterns,
- *String*/0: strings and substring patterns (contains, starts with, ...),

Constructors define the structure of formulas and interpretations

- *Sum*/2: the disjoint union of 2 logics (both syntax and interpretation domain),
- *Prod*/2: the product of 2 logics (both syntax and interpretation domain),
- *Multiset*/1: multisets over formulas of the sub-logic,

Abstractors extend the abstract syntax without changing the interpretation domain

- *Top*/1: adds a (complete) top to the sub-logic,
- *Interval*/1: intervals over values taken as the formulas of the sub-logic,
- *Prop*/1: closure of the syntax by the 3 boolean connectors (and, or, not), the combinations of these connectors in *TELL*-formulas is constrained,

Adaptors help to ensure some properties are satisfied

- *Set*/1: applies the powerset to the interpretation domain; it reduces the need for the property *reduced* to the weaker property *reduced.right*,
- *Bottom*/1: adds a bottom to the logic, and extends the definition of its conjunction; it replaces the need for the property *reduced'* by the property sg',
- *Single*/1: applies on formulas the epistemic modalities of the logic AIK [Lev90]; it enforces the property sg' (*TELL*-formulas have a single model), and helps to introduce the Closed World Assumption (see Section 4.2.3).

Given some logics $L_1, ..., L_n$, the type of the composed logic $L = F(L_1, ..., L_n)$ is computed by T_F from $T_{L_1}, ..., T_{L_n}$ (Definition 5). Concretely T_F is a set of theorems relating the properties of the logics $L_1, ..., L_n$ to the properties of the logic L. These theorems have all the form

$$property\ of\ L \Leftarrow conjunction\ of\ properties\ of\ L_1, ..., L_n.$$

Similarly, properties on logics are considered as type assignments, L_i : *properties*, so that proving that L has some property is simply to type-check it. This type-checking problem can be represented by a set of (possibly recursive) equations on types, which is solved by classical techniques of fixpoint iteration on finite domains [DP90]. The type-checking is performed automatically by the Composer. The possible recursivity comes from the fact that a logic can be defined in a recursive way, as illustrated in the next Section. More details on this type-ckecking are given in Section 2.4.

2.4 Type Checking

If a logic can be defined in a non-recursive way, then it can be seen as a tree. Its nodes and leaves are logic functors, and the arity of each node is the arity of the logic functor. Each sub-tree corresponds to a sub-logic. The type of the built logic can then easily be synthesized starting from the type of the leaves, which are both logic functors and logics, and going upward, each functor computing the type of the sub-logic it defines from the type of its argument logics. This means that every logic functor F is equiped with a function mapping the types of its arguments to the type of the built sub-logic. This function is the concretization or implementation of the type T_F of the logic functor. We will abuse notation by using T_F both as the type and the function. In the case a logic functor expects no argument, this function reduces to a constant type.

The difficulty comes with recursive definitions as in such a case we obtain a rooted graph instead of a tree, where the root is the leftmost logic functor in the definition. This root is the only entry point of the graph.

Each functor F_i used in the definition of a logic defines a sub-logic L_i according to the equation

$$L_i = F_i(L_{i_1}, ..., L_{i_n}).$$

According to Definition 5, this entails an equation on the type of sub-logics:

$$T_{L_i} = T_{F_i}(T_{L_{i_1}}, ..., T_{L_{i_n}}).$$

The types of all sub-logics can be gathered in a type $T = (T_{L_i})_{i \in I}$, which is a function from sub-logic indices to logic types $(I \to \mathbb{T})$. Then type functions T_{F_i} can be adapted to apply on the global type T, and gathered in a global type function T_F, such that

$$T_F(T)(i) = T_{F_i}(T_{i_1}, ..., T_{i_n}),$$

where n is the arity of F_i, and $i_1, ..., i_n$ are the indices of the sub-logics that are the arguments of the functor F_i.

The function T_F is a map on the domain defined by the type $(I \to \mathbb{T})$, which can be ordered in the following way.

Definition 6 (order on global types) *Let $S, T \in (I \to \mathbb{T})$ be global types. We say S contains T, and we note $S \sqsubseteq_T T$, iff for all $i \in I$, we have $S(i) \supseteq T(i)$. The maximal global type according to this order is the function that maps every index i to the full set of logic properties, which is denoted by $\bot_T$.*

Now, as every property on every sub-logic is equated to a conjunction of properties on other sub-logics, it follows that the function T_F is order-preserving w.r.t. the ordering $\sqsubseteq_T$. Indeed, suppose $S \sqsubseteq_T T$. This means every property present in T is also present in S. So, if a property p is present in $T_F(T)$, then this implies all properties p depends on are in T, and so are also in S. Hence, the property p is also in $T_F(S)$. Hence $T_F(S) \sqsubseteq_T T_F(T)$. With the fact that the domain of global types is finite, we can conclude that the fixpoint of T_F is defined, and can easily be computed by applying iteratively T_F on the maximal global type $\bot_T$ until reaching the fixpoint $\mu(T_F)$ [DP90]. The type of the built logic is then defined by $\mu(T_F)(0)$, where 0 is the index of the root functor.

Chapter 3

Implementation as ML Functors

Logic functors are not only formally specified, but also implemented as modules in a programming language so that they can easily be composed to form various executable logics. In particular logic functors are very close to *parameterized modules* [Mac88a], where modules play the role of logics as a combination of an abstract syntax and a set of operations over this syntax, and where parameters play the role of logics as arguments of a logic functor. These parameterized modules are available in some ML languages, and we use Objective CAML (OCaml[1]), where they are called *functors*. OCaml is a functional programming language that offers a rich and safe type system, modules, (module) functors, and even recursive module definitions.

We first define the module type, or signature, that mimicks our definition of a logic (Section 3.1). Then we show how easy it is to compose logic functors directly in the OCaml language, without having to make a dedicated composer (Section 3.2). We also explain how the type of built logics is automatically computed from the type of logic functors (Section 3.3). Finally we introduce some pratical aspects of the implementation of logic functors, which were not relevant at the theoretical level (Section 3.4).

3.1 Types and Signatures

A logic is defined as a structure $L = (AS, S, P, T)$, where AS is the abstract syntax, S is the semantics, P is the set of operations, and T is the type (Definition 4). All these elements, except the semantics, are represented in the following signature.

```
module type T =
  sig
    type t
```

[1] See `http://caml.inria.fr/`.

```
    val tell : t -> bool
    val ask : t -> bool

    val subs : t -> t -> bool
    val top : unit -> t option
    val bot : unit -> t option
    val conj : t -> t -> t option
    val disj : t -> t -> t list
    val le_l : t -> t -> bool option
    val le_u : t -> t -> bool option

    val props : unit -> props
  end
```

`T` is the name of the signature, or module type. It is composed of a type `t`, and a set of values (keyword `val`). The type `t` stands for the abstract syntax AS as it specifies a set of values, the abstract syntax trees (or internal representation) of formulas. The values stands for operations P, except the last one, `props`, which stands for the type T of the logic, i.e., the set of properties satisfied by the logic.

The value `props` is a function that returns this set of properties. It is a function as the type of a built logic must be computed. This set of properties is represented as being of type `props`, which is defined below.

```
type requirements = string list

type props = {
    st : requirements;
    st' : requirements;
    sg' : requirements;
    cs_subs : requirements;
    cp_subs : requirements;
    cp'_subs : requirements;
    cp_top : requirements;
    cs_bot : requirements;
    defst_conj : requirements;
    cs_conj : requirements;
    cp_conj : requirements;
    cs_disj : requirements;
    cp_disj : requirements;
    cs_le_l : requirements;
    cp_le_l : requirements;
    cs_le_u : requirements;
    cp_le_u : requirements;
```

```
    reduced : requirements;
    reduced' : requirements;
    reduced_top : requirements;
    reduced_bot : requirements;
    reduced_right : requirements;
  }
```

A set of properties is represented as a record, where each slot corresponds to a property, and each value associated to a slot is a list of requirements for this property to be true. One could think a boolean would be enough to tell whether a property is satisfied or not. However it appears in practice that it is very useful to know why a property is not satisfied. In a built logic the satisfaction of a property is a function of the satisfaction of some properties in logic functors. More concretely, a property is satisfied if a set of properties over some logic functors are all satisfied. So, the value associated to a property is the subset of these properties that are not satisfied. If this subset is empty, then the property is satisfied. Otherwise, this means that at least one required property is not satisfied. There may be two reasons for this:

- either this required property is proved false, and then a new composition of functors has to be imagined in order to satisfy the property in the built logic,
- or there is no theorem about the required property in the type of its logic functor, and one has to be found and proved.

Examples of the building of a logic, and its adaptation to make it satisfy some properties are given in Section 4.

The value `tall` (resp. `ask`) is a filter over the abstract syntax, and enables us to test whether a formula $f \in AS$ is a *TELL*-formula $f \in TELL$ (resp. a *ASK*-formula $f \in ASF$).

The values `subs`, `top`, `bot`, `conj`, `disj`, `le_l`, `le_u` stand respectively for subsumption, tautology, contradiction, conjunction, disjunction, lower ordering, and upper ordering. Their OCaml type is a direct translation from Section 2.2, with the following details:

- the type constructor `option` is used when the operation can return *undef*,
- the type constructor `list` is used to represent sets like in disjunction,
- constants like top and bottom are made functions (by adding a `unit` argument) as this is required by OCaml in order to build recursive modules.

It can be seen that this interface can easily be extended with new operations, and new properties. A default definition of a logic is given below. It is useful when defining logic functors where not all operations are defined, but which still have to match the logic signature `T`.

```
module Default : T =
  struct
```

```
    let tell f = true
    let ask f = true

    let subs f g = false
    let top () = None
    let bot () = None
    let conj f g = None
    let disj f g = [f; g]
    let le_l f g = None
    let le_u f g = None

    let props () = no_props "Default"
  end
```

The function `no_props` F, where F is the name of a logic functor, returns a record of properties where each slot is in the form $p = [\text{"}F.p\text{"}]$, that is no property is satisfied.

Now we have presented the signature of a logic, and its default implementation, we give a schematic example of the implementation of a logic functor F as a parameterized module.

```
module F (L : T) : T =
  struct
    include Default

    type t = ... L.t ...

    let subs f g = ... L.subs ... L.top ...
    ...

    let props () = ... L.props ...
  end
```

`F` is the name of the functor. It takes one logic with signature `T` as a parameter, and returns another logic also with signature `T`. The returned module implements the type and values of the signature `T` as a function of the type and values of the argument logic `L`. It includes the logic `Default` in order to ensure that the signature is fully implemented, even if not all operations are explicitly defined.

Many examples of implemented logic functors are available in the LogFun library. The module `logic.ml` contains types and signatures, as well as the default logic, and a logic tester. There are only slight variations with definitions above, such as operation names, and the use of exceptions instead of options for partially defined operations. These variations comes only from historical reasons, and are only a different style of programming.

3.2 Composition of Logic Functors

While implementing a logic functor requires both logic and programming knowledge, composing them for building customized logics is accessible to application designers, and does not even require knowledge of the programming language OCaml as shown in the following.

Let us first assume we have a few logic functors at disposal. `Atom` expects no argument and corresponds to the usual atoms that are found in many logics. `String` represents the concrete domain of strings, and patterns on strings (e.g., "starts with", "contains"). `List` is used to represent patterns on lists, whose elements belong to its argument logic `L` (in fact, `List` is a combinator of smaller functors, as can be seen in the appendices). Finally, `Prod` builds the product of 2 logics `L1`, and `L2`; so, its formulas are all made of a formula from `L1` and a formula from `L2`.

```
module Atom : T = ...
module String : T = ...
module List (L : T) : T = ...
module Prod (L1 : T) (L2 : T) : T = ...
```

These logic functors can now be composed in a functional way, according to their respective arity. A logic is then defined as a OCaml module, given a name and a composition of logic functors.

```
module L1 : T = List (Prod Atom String)
module L2 : T = Prod (List Atom) (List String)
module rec L3 : T = Prod (Prod Atom String) (List L3)
```

The logic `L1` enables us to represent and reason on lists of couples, each couple being composed of an atom and a string pattern, i.e., each couple is a string-valued attribute. In logic `L2` the order of application of `List` and `Prod` is reversed so that now formulas are the combination of a list of atoms, and a list of string patterns. The logic `L3` is an example of a recursively defined logic, as it is used in its own definition. In such a case one has to be cautious the recursive definition is well-founded. This is the case here as a list can be empty, which allows for finite formulas, and makes operations terminate. Formulas in `L3` are the combination of a string-valued attribute, and a list of formulas in `L3`. So, these formulas represent trees, whose nodes are labeled by string-valued attributes. It is useful to define such a tree structure once for all kind of node labels. This is made possible by defining a new logic functor as a combination of existing logic functors, which is called a *combinator*. This combinator can then be used as any other logic functor. In some way, it is only syntactic sugar, but it improves abstraction and reusability of logic functors.

```
module Tree (Label : T) (Rec : T) : T = Prod Label (List Rec)
module rec L3 : T = Tree (Prod Atom String) L3
```

OCaml functors cannot be defined in a recursive way, so that an additional argument `Rec` is needed in the definition of `Tree`. The representation of trees can be abstracted further

by replacing the logic functor `List` by an additional parameter telling how the children of a node are organized.

```
module Tree (Children : functor (C : T) -> T) (Label : T) (Rec : T) : T =
    Prod Label (Children Rec)
```

From this new combinator, it is straightforward to define binary or n-ary trees, simply instantiating the parameter `Children` by `PairOrNil` (another combinator) or `List`.

```
module PairOrNil (X : T) : T = Sum (Prod X X) Nil
module BinTree = Tree PairOrNil
module NaryTree = Tree List
```

3.3 Automatic Type Checking of Built Logics

The type checking of built logics, i.e., the computation of its set of properties, is simply done by evaluating the function `props` defined in the signature of logics. This function is coded in each functor F as a translation of its theorems T_F, according to the type `props`.

In the case of an atomic functor F, the set of properties is simply defined as a record where each property-slot is given a value among `isok` (empty list of requirements), and `requires F p` (list of one requirement "F.p", the property p is required on the functor F). We give below the example of the functor *Atom*, where the expression `no_props "Atom"` returns a default set of properties which are all required. Then only satisfied properties need to be explicitely stated.

```
let props () =
  {no_props "Atom" with
   st' = isok;
   sg' = isok;
   cs_entails = isok;
   cp_entails = isok;
   cp'_entails = isok;
   cp_top = isok;
   cs_bot = isok;
   defst_conj = isok;
   cs_conj = isok;
   cp_conj = isok;
   cs_disj = isok;
   cp_disj = isok;
   reduced_right = isok;
  }
```

In the case of a non-atomic functor, the set of properties is usually defined as a function of the properties of its logic arguments. Let us consider the example of the logic functor *Sum*, and one of its theorems:

$$sg'_{\sqsubseteq} \Leftarrow sg'_{\sqsubseteq_1} \wedge sg'_{\sqsubseteq_2}.$$

With the utilitary function `reqand` that encodes the union of sets of requirements, this theorem can be directly translated to

```
sg'_subs = reqand [a1.sg'_subs; a2.sg'_subs]
```

.

Now a problem comes from the fact that logics can be defined in a recursive way. If no care is taken, this will result in an infinite loop, and then non-termination of the computation of properties. So, we use a universal top-down fixpoint algorithm [LCVH92]. We adapted it as a higher-order function so that it can be applied locally in each logic functor, without requiring any global data-structure. We systematically apply it in the definition of the function `props` in non-atomic functors.

```
let fixpoint : (unit -> props) -> (unit -> props) =
  fun p ->
    let rec f : unit -> props =
      let l = ref all_props in
      let is_ancestor = ref false in
      let was_visited = ref false in
      fun () ->
        if !is_ancestor (* loop detection *)
        then begin (* stop recursion *)
          was_visited := true;
          !l end
        else begin
          was_visited := false;
          is_ancestor := true;
          let l' = p () in (* evaluation of properties *)
          is_ancestor := false;
          if not !was_visited or l' = !l (* stable value for props *)
          then !l
          else begin
            l := l';
            f () end (* recursive call *)
        end in
    f
```

The definition for the functor *Sum* can now be given in details.

```
let props =
  fixpoint
    (fun () ->
      let l1 = L1.props () in
      let l2 = L2.props () in
      {no_props "Sum" with
       st' = reqand [l1.st'; l2.st'];
       sg' = reqand [l1.sg'; l2.sg'];
       cs_subs = reqand [l1.cs_subs; l1.cs_bot;
                         l2.cs_subs; l2.cs_bot];
       cp_subs = reqand [l1.cp_subs; l1.reduced_bot;
                         l2.cp_subs; l2.reduced_bot];
       cp'_subs = reqand [l1.st'; l1.cp'_subs;
                          l2.st'; l2.cp'_subs];
       cp_top = isok;
       cs_bot = isok;
       cs_conj = reqand [l1.cs_conj; l2.cs_conj];
       cp_conj = reqand [l1.cp_conj; l2.cp_conj];
       cs_disj = reqand [l1.cs_disj; l2.cs_disj];
       cp_disj = reqand [l1.cp_disj; l2.cp_disj];
       reduced = reqand [l1.reduced; l2.reduced];
       reduced_top = reqand [l1.reduced_top; l2.reduced_top];
       reduced_bot = reqand [l1.reduced_bot; l2.reduced_bot];
       reduced_right = reqand [l1.reduced_right; l2.reduced_right];
      })
```

3.4 Practical Aspects of Logic Functors

We must add to the above operations of an implementation, a parser and a printer for handling the concrete syntax of formulas. Indeed, an application should input and output formulas in a readable format. For parsing formulas we use the stream parsers available in OCaml. They are limited to LL_1 grammars but can easily be modularized inside logic functors.

It is possible to customize the concrete syntax by passing to functors an additional parameter `Param`. It is a small non-logical module containing some functor-specific values like tokens to be used as separators, for instance. These extra-parameters can also be used to offer several versions of a logic functor in only one module.

It is also possible to add other operations to logics. For instance, we already defined operations like a refinement operator for machine learning [FK05], or a feature extractor for navigation in logical information systems [FR04].

Chapter 4

Examples

This chapter aims to give a more concrete feeling of the use of logic functors. So, we here adopt the point of view of the Application Designer, which have access to a toolbox made of all the logic functors defined in appendices, and coded in Objective Caml as described in Chapter 3. The more relevant aspects of logic functors for the Application Designer are: the syntax, the operations that are defined, and the properties that are satisfied, and under which conditions.

4.1 A Logic for Logical Information Systems

In the introduction of this document, we introduced as a motivating example the description and querying of bibliographic references. In this section, we progressively develop a logic for representing both descriptions and queries. In each version, we give a definition of the logic as a module `L`, the definition of a description `descr`, and the definition of a typical query `query`.

As a starting point, let us consider that each reference can be described by a set of simple attributes. These attributes can be represented as *atoms* (logic functor `Atom`)), and sets of attributes can be represented by multisets of atoms (logic functor `Multiset`), as multisets are more general than sets. Hence the first definition of a logic for descriptions.

```
module L = Multiset Atom
descr: {TheJungleBook, paper-back, recent}
query: {recent, TheJungleBook}
```

Now we would like to have valued attributes in descriptions. Values can be either strings, or integers (logic functors `Sum`, `String`, `Int`); and valued attributes can be built as the product of an atom and a value (logic functor `Prod`). But we need to make the value optional in order to retain simple attributes in descriptions (logic functor `Option`).

```
module Val = Option (Sum String Int)
module L = Multiset (Prod Atom Val)
descr: {title is "The Jungle Book", paper-back, year = 1985}
query: title contains "Jungle"
query: {paper-back, year = 1985}
```

In the latter logic queries are limited in that one cannot express intervals over integers, or the presence of an attribute without specifying its type (simple, string-valued, or integer-valued). This can be solved by applying the abstractors `Interval`, and `Top` where appropriate in the definition of the logic.

```
module Val = Top (Option (Sum String (Interval Int)))
module L = Multiset (Prod Atom Val)
descr: {title is "The Jungle Book", paper-back, year = 1985}
query: title ?
query: {paper-back, year in 1980 .. 1990}
```

The expressivity of queries is still limited in that disjunction and negation cannot be expressed, and conjunction only applies to different elements of a description. In order to allow full boolean queries, both on description elements and whole descriptions, we apply twice the abstractor `Prop`: internally and externally to `Multiset`. As the syntax in functors can be parameterized (not detailed here), we can distinguish internal connectors from external connectors: external connectors are `and, or, not, implies`, and internal connectors are `&, |, !, ->`.

```
module Val = Top (Option (Sum String (Interval Int)))
module L = Prop (Multiset (Prop (Prod Atom Val)))
descr: {title is "The Jungle Book", paper-back, year = 1985}
query: (title contains "Jungle" & ! title contains "Wood") and
       (paper-back or not (year in 1970 .. 1980 | year in 1990 .. 2000))
```

We now have a convenient logic w.r.t. the expressivity of descriptions and queries. Moreover it is very easy to add value types by simply extending the definition of the sub-logic `Val` with other concrete domains. But before using this logic for information retrieval we must ensure it satisfies the logic properties `cs_subs` and `cp'_subs`. If we call the function `L.props ()` in order to compute the properties of `L`, we discover that consistency is satisfied, but that completeness requires the property `reduced'` on the sub-logic starting at `Multiset`. It happens that the logic functor `Bottom` helps to ensure this property, so we can try and apply it on top of `Multiset`.

```
module Val = Top (Option (Sum String (Interval Int)))
module L = Prop (Bottom (Multiset (Prop (Prod Atom Val))))
  cs_subs requires Prod.reduced' Multiset.sg'
  cp'_subs requires Prod.reduced' Multiset.sg'
```

Now we get that the property `sg'` is required on `Multiset`, and the property `reduced'` is required on `Prod`. For the latter case, we can apply the same solution as above (logic functor `Bottom`). For the former case, it happens the logic functor `Single` ensures the property `sg'`, so we apply it on top of the functor `Prod`.

```
module Val = Top (Option (Sum String (Interval Int)))
module L = Prop (Bottom (Single (Multiset (Prop (Bottom (Prod Atom Val))))))
  cs_entails is ok
  cp'_entails is ok
descr: [{title is "The Jungle Book", paper-back, year = 1985}]
query: (title contains "Jungle" & ! title contains "Wood") or
       paper-back and not (year in 1970 .. 1980 | year in 1990 .. 2000)
```

We finally have a well-defined logic, with a consistent and (partially) complete subsumption. Note how the user is guided in the process of defining its logic as missing requirements are provided for every property that is not satisfied. The syntax of descriptions is slightly changed as the logic functor `Single` requires to square-bracket formulas occuring in descriptions.

4.2 The Reconstruction of ALC-like Description Logics

Description Logics (DL) are widely used in knowledge representation and information systems [CLN98]. Like our logics, they are based on a notion of subsumption $\sqsubseteq$, and in fact our Initiators can be seen as an impoverished variant of DL. So, an interesting question is to know whether logic functors can be used to reconstruct description logics ? We show in this section that the subsumption test of $\mathcal{ALC}$ (without TBox) can be nicely reconstructed. We then generalize it as a combinator, which makes it easy to specialize $\mathcal{ALC}$ to concrete domains, more expressive roles, and Closed World Assumption.

The abstract syntax of the logic $\mathcal{ALC}$ is defined by:

$$C \to A \mid \texttt{some}\ r\ C \mid \texttt{all}\ r\ C \mid \texttt{not}\ C \mid C\ \texttt{and}\ C \mid C\ \texttt{or}\ C \mid \texttt{any} \mid \texttt{none},$$

where A, and r respectively stand for *atomic concepts*, and *roles*. These are properly represented by atoms (logic functor *Atom*). In the semantics of $\mathcal{ALC}$ a formula `all` r C is equivalent to the formula `not some` r `not` C, so that we can restrict ourselves to existential quantification. When some object satisfies a formula `some` r C, this means it is related to another object satisfying C, through a relation satisfying r. So the property `some` r C can be represented as the product (r, C) of a role, and a complex concept (logic functor *Prod*): its concrete syntax can easily be customized as (`some` r C) for readability[1]. All other connectors of the language are provided by the logic functor *Prop*. This leads to the following first attempt of defining the logic $\mathcal{ALC}$ with logic functors.

[1] Every logic functor comes with a parser and a printer for the concrete syntax, which can easily be customized. Syntactic sugar for the quantifier `all` can also be obtained.

```
module rec ALC : T = Prop (Sum Atom (Prod Atom ALC))
  cs_subs is ok
  cp_subs requires Prod.reduced Atom.reduced
```

The Composer produces a new logic `ALC` from its defining expression. Note the way it is recursively defined in order to account for complex concepts inside quantifiers. The Composer also performs its type-checking, whose result says the subsumption is proved consistent, but not complete because the property *reduced* could not be proved on the sub-logics built by the functors *Prod*, and *Atom* (both arguments of the functor *Sum*).

The above definition is correct about the syntax, but what about the interpretation domain ? According to the definition of functors (see Appendices), it is recursively defined as

$$I = Atom \uplus (Atom \times I),$$

i.e., either a primitive concept (an atom), or a pair made of a primitive role (an atom), and another interpretation. This is not satisfactory because an $\mathcal{ALC}$-interpretation should not be a single atomic concept or single role, but should be a set of atomic concepts and roles. This adaptation to the interpretation domain can be obtained by the logic functor *Set*, whose side effect is to reduced the need for the property *reduced* to the weaker property *reduced.right*.

```
module rec ALC : T = Prop (Set (Sum Atom (Prod Atom ALC)))
  cs_subs is ok
  cp_subs is ok
descr: tall and
       (some child male) and
       not (some child not tall)
query: (some child male and tall) and
       not (some child not (male implies tall))
```

This time the interpretation domain is $I = \mathcal{P}(Atom \uplus (Atom \times I))$, and the resulting subsumption prover is proved both consistent and complete by the Composer. In the formulas, the keyword `some` represents the existential quantifier. The example description d tells someone is tall, has a male child, and has only tall children. The example query q asks whether somebody has a tall male child, and whose male children are all tall. It can be proved in ALC that the description is subsumed by the query: $d \sqsubseteq_{ALC} q$.

We have showed our logic `ALC` has the same syntax as $\mathcal{ALC}$, a satisfactory semantics, and proved properties w.r.t. the subsumption prover. However it remains to show whether it is semantically equivalent to the description logic $\mathcal{ALC}$? In description logics, the semantics of the subsumption relation $f \sqsubseteq g$ is that, for all interpretation $\mathcal{I} = (\Delta_\mathcal{I}, .^\mathcal{I})$, we have $f^\mathcal{I} \subseteq g^\mathcal{I}$, where $f^\mathcal{I}$ denotes the set of instances of f among the individual domain $\Delta_\mathcal{I}$. This inclusion can be successively rewritten as

$$\begin{array}{l} \forall \mathcal{I} : \forall i \in \Delta_\mathcal{I} : i \in f^\mathcal{I} \Rightarrow i \in g^\mathcal{I}, \\ \forall i' = (\mathcal{I}, i) : i' \models f \Rightarrow i' \models g, \\ M'(f) \subseteq M'(g) \quad \text{where } M'(f) = \{(\mathcal{I}, i) \mid i \in f^\mathcal{I}\}. \end{array}$$

The last proposition is similar to our semantics of subsumption, as defined by the properties $cs_{\sqsubseteq}$, and $cp_{\sqsubseteq}$, except it applies on a different domain of interpretations $I' = \{(\mathcal{I}, i) \mid i \in \Delta_{\mathcal{I}}\}$. This implies that if we can find a pair of maps $\phi : I' \to I$, and $\phi' : I \to I'$ that preserve the satisfaction relation $\models$ between interpretations and formulas, then the two interpretations can be said equivalent, and `ALC` equivalent to $\mathcal{ALC}$ for the subsumption test.

Definition 7 *Let $I = \mathcal{P}(Atom \uplus (Atom \times I))$ be the interpretation domain of ALC, and let $I' = \{(\mathcal{I}, i) \mid i \in \Delta_{\mathcal{I}}\}$ be the interpretation domain of $\mathcal{ALC}$ w.r.t. subsumption. A first map from I' to I is defined by*

$$\begin{array}{rcl} \phi((\mathcal{I}, i)) & = & \{A \in Atom \mid i \in A^{\mathcal{I}}\} \\ & \cup & \{(r, j) \in Atom \times I \mid \exists i' \in \Delta_{\mathcal{I}} : (i, i') \in r^{\mathcal{I}} \wedge j = \phi((\mathcal{I}, i'))\}. \end{array}$$

A second map from I to I' is defined by

$$\begin{array}{rcl} \phi'(i) & = & (\mathcal{I}(i), i) \\ & s.t. & \Delta_{\mathcal{I}(i)} = \{i\} \cup \bigcup\{\Delta_{\mathcal{I}(j)} \mid (r, j) \in i\} \\ & & A^{\mathcal{I}(i)} = \{j \in \Delta_{\mathcal{I}(i)} \mid A \in j\}, \textit{for all } A \in Atom \\ & & r^{\mathcal{I}(i)} = \{(j, k) \in \Delta_{\mathcal{I}(i)} \times \Delta_{\mathcal{I}(i)} \mid (r, k) \in j\}, \textit{for all } r \in Atom. \end{array}$$

The idea behind these maps is that I-interpretations are trees, where nodes are labelled by sets of atomic concepts, and edges are labelled by roles. $\mathcal{I}$-interpretations can be arbitrary graphs instead of trees but, in the subsumption test, only one individual i_0 is considered at a time, only individuals accessible from i_0 are taken into account, and there is no way in $\mathcal{ALC}$ to say that 2 roles point to the same individual. Hence I'-interpretations can also be seen as trees, and we obtain the following theorem.

Theorem 8 *For every formula in $AS_{ALC} = AS_{\mathcal{ALC}}$, we verify*

$$\begin{array}{l} i \models f \iff \phi'(i) \models f, \textit{for all } i \in I \\ i' \models f \iff \phi(i') \models f, \textit{for all } i' \in I' \end{array}$$

The proof is obtained by induction on the structure of formulas, and the interesting cases are the atomic and existentially quantified concepts.

Variations on the logic ALC A major advantage of logic functors is that, starting from a composed logic, it is easy to adapt or extend it as it only requires changing the definition of the logic, and type-checking the result. This is in contrast with the usual approach where most often such an extension requires a full paper for presenting the new syntax, semantics, subsumption algorithm, and correctness proofs. We present in the following 3 extensions of the logic `ALC`: (1) concrete domains in addition to atoms, (2) complex roles (negation, conjunction, and disjunction in roles), and (3) application of the Closed World Assumption on descriptions. All these extensions are defined in one line, and require no proof from the user as they are performed automatically by the Composer.

In order to factorize these extensions, we define a combinator that define once for all the general structure of the logic ALC.

```
module GenALC (A : T) (R : T) (Rec : T) : T = Prop (Set (Sum A (Prod R Rec)))
module rec ALC : T = GenALC Atom Atom ALC
```

Unfortunately, in OCaml, recursivity cannot be used when defining a functor, but only when defining a module. This is why the additional argument `Rec` appears in the definition of `GenALC`. Then, when defining a logic, this argument is simply replaced by the name of the logic being defined.

4.2.1 Concrete Domains

In many applications, it can be useful to have concrete domains, such as integers or strings, in addition to the abstract atoms [HS01].

```
module rec ALC1 : T = GenALC (Sum Atom (Sum String Int)) Atom ALC1
  cs_subs is ok
  cp_subs is ok
descr: tall and
       (some name is "Peter") and
       (some age 39) and
       (some child (some name is "Arthur") and (some age 16))
query: (some name contains "Peter" or contains "Paul") and
       (some child (some age 20) or (some name any))
```

In `ALC1`, atomic concepts can now be strings, substrings, and integers in addition to classic atoms. The subsumption operation of this logic is automatically proved consistent and complete by the composer. The description d tells somebody is tall, has a name "Peter", is aged 39, and has a child, who has a name "Arthur", and is aged 16. The query q asks for people whose name contains "Peter" or "Paul", and who has a child, who has a name or whose age is 20. It can be proved that $d \sqsubseteq_{ALC1} q$.

4.2.2 Complex Roles

In previous logics, boolean operations only apply on concepts, not on roles. However papers in the litterature consider the use of negation, conjunction, and disjunction on roles [LS00]. The conjunction enables us to tell that 2 objects are related by 2 kinds of relations (e.g., friend *and* cousin). The disjunction helps to express undetermined or general relation (e.g., like in "father or mother is a doctor"). The negation solves the problem of expressing axioms like "Mary likes all cats" [LS00], which can be represented by `all (not likes) not cat`, and is opposite to `all likes cat` ("Mary likes only cats").

```
module rec ALC2 : T = GenALC (Sum Atom (Sum String Int)) (Prop (Set Atom)) ALC2
  cs_subs is ok
  cp_subs is ok
descr: (some (cousin and friend) doctor and (some name is "Jack")) and
```

```
       not (some (not likes) cat) and
       (some has cat)
query: (some (likes or friend) doctor) and
       (some likes cat)
```

In `ALC2` (which extends `ALC1`), roles can now be arbitrary boolean combinations of atoms. The subsumption operation of this logic is automatically proved consistent and complete by the composer. The description d tells somebody has some cousin and friend, who is a doctor and has name "Jack", likes all cats, and has some cat. The query q asks for people, who likes a doctor, or has a doctor as a friend, and who likes a cat. It can be proved that $d \sqsubseteq_{ALC2} q$.

4.2.3 Closed World Assumption

Let us consider the following description and queries in the above logic `ALC1`.

```
descr: (some name is "Peter") and not female
query: not female
query: not (some name is "Arthur")
```

The description is subsumed by the first query, but not by the second. This is annoying because in information systems, we like the ability to retrieve all objects not satisfying some property. Here, we get a answer to a negative query, only when this negation is explicitly stated in the description. This is known as the "Open World Assumption", as opposed to the "Closed World Assumption", which is usually desired in information systems. Under the latter setting, everything not said in the description is considered as false. This is usually realised through non-monotonic reasoning, which has already been introduced in description logics [DNR97]

In fact, it is possible to combine explicit and implicit negation in the same monotonic logic by using the epistemic logic *AIK* [Lev90]. The principle is to bracket descriptions with a modality meaning "All I know", and query atoms with a modality meaning "I know". We have generalized this principle as the *epistemic extension*, which can then be applied on various logics [Fer06]. This epistemic extension is realized through the logic functor *Single*, whose neat result is to enforce the property sg' (hence its name).

So, we first apply the functor *Single* on the logic *ALC1*. Then, as we need negation both inside and outside the scope of modalities, we apply again the functor *Prop* on the result. Finally we insert the logic functor *Bottom* in order to get the right properties on the subsumption operation.

```
module rec ALC3 : T = Prop (Bottom (Single ALC1))
  cs_subs is ok
  cp'_subs is ok
descr: [(some name is "Peter") and not female]
query: !(not female)
query: not !(some name is "Arthur")
```

Note we loose the full completeness of subsumption, but we still have completeness for subsumption between *TELL*-formulas and *ASK*-formulas, which is what counts in information systems. The square brackets denote the modality "All I know", and the exclamation mark denotes the modality "I know". The first query can be read "I know he is not a female", and second can be read "I do not know he has name Arthur". Both queries now subsume the description, which solves our problem. If we further assume we have a complete knowledge, i.e. everything I do not know is false, then the external negation can be read as usual negation, and the internal negation can be read as *opposition*. Then the first query reads "he is male (the opposite of female)", and the second query reads "he has not name Arthur".

Chapter 5

Conclusion

The framework of logic functors makes it possible for an end-user to design safely a new logic. A theorem prover and its consistency/completeness proof are derived automatically. If necessary the proof exhibits prerequisites that may indicate how to build more consistent/complete variants of the logic. The framework is implemented in CAML, and uses in a crucial way the capability of the ML family to develop parameterized modules (also called functors in that domain). The framework occupies an original position between plain programming in which almost nothing semantic can be proved, and option selection that leaves very little choice [dCFG+01]. We call this original position "gluing".

The semantics of logic functors is given in a style that is close to the semantics of description logics. We have shown how to rebuild the existing logic $\mathcal{ALC}$, and how to design new versions of it. However, the semantics style of logic functors does not limit applications to variants of description logics. We have also designed a logic for Logical Information Systems that is based on multisets, valued attributes, concrete domains, boolean operators, and the Closed World Assumption.

Our paper suggests a software architecture for logic-based systems, in which the system is generic in the logic, and the logic component can be separately defined, and plugged in when needed. We have realized a prototype Logical Information System along these lines [FR04]: Camelis, which can be freely downloaded at `http://www.irisa.fr/lande/ferre/camelis`.

5.1 Related work

Our use of the word *functor* is similar to ML's one for designating *parameterized modules* [Mac88b], and in fact our logic functors *are* ML modules. However, our logic functors are very specialized contrary to functors in ML which are general purpose (in short, we have fixed the signature), and they carry a semantic component. Both the specialization and the semantic component allow us to express composition conditions that are out of the scope of ML functors.

The theory of *institutions* [GB92] shares our concern for *customized logics*, and also uses the word *functor*. However, the focus and theoretical ground are different. Institutions focus on the relation between notations and semantics, whereas we focus on the relation between semantics and operations. In fact, the operations class $\mathbb{P}$ is necessary for us to enforce *embeddability*. The theory of institutions is developed using category theory and in that theory there are functors from signatures to formulas, from signatures to models, and from institutions to institutions. Our logic functors correspond to parameterized institutions.

An important work which shares our motivations is LeanTAP [BP95]. The authors of LeanTAP have also recognized the need for embedding customized logics in applications. To this end, they propose a very concise style of theorem proving, which they call *lean theorem proving*, and they claim that a theorem prover written in this style is so concise that it is very easy to modify it in order to accomodate a different logic. And indeed, they have proposed a theorem prover for first-order logic, and several variants of it for modal logic, etc. Note that the first-order theorem prover is less than 20 clauses of Prolog. However their claim does not take into account the fact that the average user is not a logic expert. There is no doubt that modifying their first-order theorem prover was easy for these authors, but we also think it could have been undertaken by few others. A hint for this is that it takes a full journal article to revisit and justify the first-order lean theorem prover [Fit98]. So, we think lean theorem proving is an interesting technology, and we have used it to define the logic functor *Prop*, but it does not actually permit the average user to build a customized logic.

Our main concern is to make sure that logic functors can be composed in a way that ensures the validity of some properties on built logics. This led us to define technical properties that simply tell us how logic functors behave: consistency/completeness, reducedness, etc. This concern is complementary to the concern of actually implementing customized logics, e.g., in logical frameworks like Isabelle [Pau94], Edinburgh LF [HHP93], or Open Mechanized Reasoning Systems [GPT96], or even using a programming language. These frameworks allow users to implement a customized logic, but do not help users in proving the completeness and consistency of the resulting theorem prover. It is not that these frameworks do not help at all. For instance, axiomatic types classes have been introduced in Isabelle [Wen97] in order to permit automatic admissibility check. Another observation is that these frameworks mostly offer environments for interactive theorem proving, which is incompatible with the objective of building fully automatic embeddable logic components.

In essence, our work is more similar to works on static program analysis toolbox (e.g., PAG [AM95]) where authors assemble known results of lattice theory to combine domain operators like product, sets of, and lists in order to build abstract domains and derive automatically a fixed-point solver for these domains. The fact that in the most favourable cases (e.g., $Prop(A)$), our subsumption relations form (partial) lattices is another connection with these works. However, our framework is more flexible because it permits to build lattices from domains that are not lattices. In particular, the logic functor *Prop* acts as a lattice completion operator on reduced logics.

5.2 Further Work

Further work is to continue the exploration and the implementation of logics as logic functors. This will certainly lead to the development of new functors. We also plan to extend our logic functors in several ways:

- add properties about the LL_1 concrete syntax parser (e.g., non-ambiguity), and/or consider more powerful parsing techniques (LL_k, LR),
- add properties about the complexity of the operations, especially the subsumption prover,
- add operations for machine learning, and data mining (e.g., refinement operators),
- consider the representation of non tree-like formulas with the help of variables,
- add theory/terminology as a parameter of the subsumption prover.

We also plan to try and reconstruct more logics, in order to challenge our logic functors and discover new ones; and to explore new applications of them.

Bibliography

[AM95] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *Static Analysis Symp.*, LNCS 983, pages 33–50, 1995.

[AMSS98] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of boolean functions for dependency analysis. *Science of Computer Programming*, 31:3–45, 1998.

[BP95] B. Beckert and J. Posegga. leanTAP: Lean, tableau-based deduction. *J. Automated Reasoning*, 11(1):43–81, 1995.

[CLN98] D. Calvanese, M. Lenzerini, and D. Nardi. Description logics for conceptual data modeling. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, pages 229–263. Kluwer, 1998.

[CSS99] M. Codish, H. Søndergaard, and P.J. Stuckey. Sharing and groundness dependencies in logic programs. *ACM TOPLAS*, 21(5):948–976, 1999.

[dCFG+01] L. Fariñas del Cerro, D. Fauthoux, O. Gasquet, A. Herzig, D. Longin, and F. Massacci. Lotrec: The generic tableau prover for modal and description logics. In R. Goré, A. Leitsch, and T. Nipkow, editors, *IJCAR*, LNAI 2083. Springer, 2001.

[DNR97] F. M. Donini, D. Nardi, and R. Rosati. Autoepistemic description logics. In *IJCAI*, 1997.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[Fer02] S. Ferré. *Systèmes d'information logiques : un paradigme logico-contextuel pour interroger, naviguer et apprendre.* Thèse d'université, Université de Rennes 1, October 2002. Accessible en ligne à l'adresse http://www.irisa.fr/bibli/publi/theses/theses02.html.

[Fer06] S. Ferré. Negation, opposition, and possibility in logical concept analysis. In Rokia Missaoui and Jürg Schmid, editors, *Int. Conf. Formal Concept Analysis*, LNCS 3874, pages 130–145. Springer, 2006.

[Fit98] M. Fitting. leanTAP revisited. *Journal of Logic and Computation*, 8(1):33–47, February 1998.

[FK05] S. Ferré and R. D. King. A dichotomic search algorithm for mining and learning in domain-specific logics. *Fundamenta Informaticae – Special Issue on Advances in Mining Graphs, Trees and Sequences*, 66(1-2):1–32, 2005.

[FR00] S. Ferré and O. Ridoux. A file system based on concept analysis. In Y. Sagiv, editor, *Int. Conf. Rules and Objects in Databases*, LNCS 1861, pages 1033–1047. Springer, 2000.

[FR04] S. Ferré and O. Ridoux. An introduction to logical information systems. *Information Processing & Management*, 40(3):383–419, 2004.

[GB92] J.A. Goguen and R.M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[GPT96] F. Giunchiglia, P. Pecchiari, and C. Talcott. Reasoning theories - towards an architecture for open mechanized reasoning systems. In F. Baader and K. U. Schulz, editors, *1st Int. Workshop: Frontiers of Combining Systems*, pages 157–174. Kluwer Academic Publishers, March 1996.

[GW99] B. Ganter and R. Wille. *Formal Concept Analysis — Mathematical Foundations.* Springer, 1999.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *JACM*, 40(1):143–184, January 1993.

[HS01] I. Horrocks and U. Sattler. Ontology reasoning in the shoq(d) description logic. In *Int. J. Conf. Artificial Intelligence*, volume 17, pages 199–204. Lawrence Erlbaum Associates Ltd, 2001.

[IB96] V. Issarny and Ch. Bidan. Aster: A framework for sound customization of distributed runtime systems. In *16th Int. Conf. Distributed Computing Systems*, 1996.

[LCVH92] B Le Charlier and P. Van Hentenryck. A universal top-down fixpoint algorithm. Technical Report CS-92-25, Brown University, Rhode Island, 1992.

[Lev90] H. Levesque. All I know: a study in autoepistemic logic. *Artificial Intelligence*, 42(2), March 1990.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Symbolic computation — Artificial Intelligence. Springer, Berlin, 1987.

[LS00] C. Lutz and U. Sattler. Mary likes all cats. In F. Baader and U. Sattler, editors, *2000 Int. Workshop in Description Logics (DL2000)*, number 33 in CEUR-WS, pages 213–226, Aachen, Germany, August 2000. RWTH Aachen.

[Mac88a] D.B. MacQueen. An implementation of Standard ML modules. In *LISP and Functional Programming*, pages 212–223, 1988.

[Mac88b] D.B. MacQueen. An implementation of Standard ML modules. In *LISP and Functional Programming*, pages 212–223, 1988.

[MS98] K. Marriott and P.J. Stuckey. *Programming with Constraints: An Introduction.* The MIT Press, 1998.

[Pau94] L. C. Paulson. *Isabelle: a generic theorem prover.* LNCS 828. Springer, New York, NY, USA, 1994.

[Poo88] D. Poole. Representing knowledge for logic-based diagnosis. In *Int. Conf. Fifth Generation Computer Systems*, pages 1282–1290. Springer, 1988.

[SFRW98] M. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to program flow analysis. *Acta Informatica*, 35(6):457–504, June 1998.

[SM83] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, 1983.

[vRCL98] C.J. van Rijsbergen, F. Crestani, and M. Lalmas, editors. *Information Retrieval: Uncertainty and Logics. Advanced models for the representation and retrieval of information.* Kluwer Academic Publishing, 1998.

[Wen97] M. Wenzel. Type classes and overloading in higher-order logic. In E.L. Gunter and A. Felty, editors, *Theorem proving in higher-order logics*, LNCS 1275, pages 307–322. Springer-Verlag, 1997.

Appendix A

How to read appendices

Each appendix is a class of logic functors. Then each section describes a different logic functor. The section title gives the name and arity of the logic functor in the form "Name/arity". In the case of a combinator, the name is equated to a logic functor composition, where the symbol λ introduces each parameter of the combinator, and the symbol μ defines it as a fixpoint (recursive structures). In some cases a logic functor is defined as an extension of a combinator, i.e., some parts of the logics are redefined. In this case, the symbol $\supseteq$ is used instead of $=$ between the name and the combinator definition.

The following section shows the structure and contents of each section describing a functor.

A.1 Name/arity [$=/\supseteq$ combinator]

The logic functor is first introduced by a short description of its formulas, interpretations, and main roles in composing logics. The following subsections are always present unless the logic functor is equal to a combinator. Whenever a logic element (e.g., operation, property) is not given, the default definition is assumed (see Chapter 2).

A.1.1 Parameters

The parameters of the functor are given a name and a type. Most often these parameters are sub-logics, but are sometimes non-logical parameters (e.g., an alphabet for strings).

A.1.2 Syntax

The syntax of formulas AS is formally defined. *TELL*-formulas, and *ASK*-formulas may be defined as subsets of AS.

A.1.3 Semantics

The interpretation domain I and the satisfaction relation $\models$ are formally defined. In some cases, the semantics can be defined in term of M instead of the satisfaction relation $\models$. The ordering on interpretations $\leq$ may also be defined.

A.1.4 Operations

Non-default operations are here defined as algorithms. This represents the executable part of a logic functor.

A.1.5 Properties

The behaviour of properties is given in the form of a theorem relating a property and the properties of sub-logics. A theorem is always accompanied by a proof, except when it is trivial given the definition of the property and definitions of the logic functor. Some properties can be considered as true, even if no theorem is given, when it is trivially true: e.g., the conjunction and disjunction are always consistent and complete when totally undefined; or when a property is a weakening of another property that is proved: e.g., sg' is true whenever sg is true.

For the concision of proofs, we assume a good knowledge of the theory of logics and logic functors in the reading of proofs. What is emphasized in proofs is the invocation of properties over sub-logics as these invocations determine the theorems.

Appendix B

Initiators

The Initiators are 0-ary logic functors. They correspond to symbols, constants, and values in concrete domains. They can also introduce patterns on these values (e.g., in *String*). They usually satisfy many properties, and so can easily be integrated in various logics.

B.1 Unit/0 = $\{unit\}$

Minimal logic with only one interpretation, and only one formula. Useful to make optional some properties in logics. $\{unit\}$ is an equivalent notation of this functor, where *unit* can be replaced by any symbol.

B.1.1 Parameters

No parameters.

B.1.2 Syntax

There is a unique formula:
$AS =_{def} \{unit\}$

B.1.3 Semantics

domain: $I =_{def} \{unit\}$.

satisfaction: $M(unit) =_{def} \{unit\}$.

B.1.4 Operations

subsumption: $unit \sqsubseteq unit$.

tautology: $\top =_{def} unit$.

conjunction: $unit \sqcap unit =_{def} unit$

disjunction: $unit \sqcup unit =_{def} unit$

B.1.5 Properties

- *df*
- *st*
- *sg*
- $cs_{\sqsubseteq}$
- $cp_{\sqsubseteq}$
- $cp_{\top}$

- *reduced*

  Proof: Let $F, G \in AS^*$, and suppose $F \not\sqsubseteq^* G$
  $\Longrightarrow \forall f \in F, g \in G : f \not\sqsubseteq g \wedge \forall g \in G : \top \not\sqsubseteq g$
  $\Longrightarrow (F = \emptyset \vee G = \emptyset) \wedge G = \emptyset$ (because $\forall f, g \in AS : f \sqsubseteq g$)
  $\Longrightarrow \bigcap_{f \in F} M(f) = I \wedge \bigcup_{g \in G} M(g) = \emptyset$
  $\Longrightarrow \bigcap_{f \in F} M(f) \not\subseteq \bigcup_{g \in G} M(g)$. ■

B.2 Atom/0

The logic whose formulas and interpretations are the usual atoms. Each atom is interpreted by itself. It is useful to represent abstract atomic properties (as opposed to concrete properties).

B.2.1 Parameters

No parameters.

B.2.2 Syntax

AS is a set of atoms.

B.2.3 Semantics

domain: $I = AS$.

satisfaction: $i \models a =_{def} i = a$.

B.2.4 Operations

subsumption: $a \sqsubseteq b =_{def} a = b$,

conjunction:

$$a \sqcap b =_{def} \begin{cases} a & \text{if } a = b \\ undef & otherwise \end{cases}$$

disjunction:

$$a \sqcap b =_{def} \begin{cases} \{a\} & \text{if } a = b \\ \{a, b\} & otherwise \end{cases}$$

B.2.5 Properties

- *df*
- *st*

  Proof: For all $a \in AS$, $M(a) = \{a\} \neq \emptyset$. ■

- *sg*

  Proof: For all $a \in AS$, $M(a) = \{a\}$. ■

- $cs_{\sqsubseteq}$

  Proof: $a \sqsubseteq b \Longrightarrow a = b \Longrightarrow M(a) = M(b) \Longrightarrow M(a) \subseteq M(b)$. ■

- $cp_{\sqsubseteq}$

  Proof: $a \not\sqsubseteq b \Longrightarrow a \neq b \Longrightarrow a \models a, a \not\models b \Longrightarrow M(a) \not\subseteq M(b)$. ■

- $defst_{\sqcap}$

  Proof: $M(a) \cap M(b) \neq \emptyset$
  $\Longrightarrow \{a\} \cap \{b\} \neq \emptyset \Longrightarrow a = b$
  $\Longrightarrow def_{\sqcap}(a, b)$. ■

- $cs_{\sqcap} \wedge cp_{\sqcap}$

- $cs_{\sqcup} \wedge cp_{\sqcup}$

- *reduced.right*
  reduced.bot

  Proof: $M(a) \subseteq \bigcup_{b \in B} M(b)$
  $\Longrightarrow \{a\} \subseteq B \Longrightarrow a \in B$
  $\Longrightarrow \exists b \in B : a = b \Longrightarrow \exists b \in B : a \sqsubseteq b$
  $\Longrightarrow \{a\} \sqsubseteq^* B$. ■

B.3 Int/0

Formulas and models are simple integers. Each integer is interpreted by itself. Interpretations are ordered according to the usual ordering on integers.

B.3.1 Parameters

No parameters.

B.3.2 Syntax

$AS =_{def} \mathbb{Z}$

B.3.3 Semantics

domain: $I =_{def} \mathbb{Z}$

satisfaction: $i \models f =_{def} i = f$

B.3.4 Operations

subsumption: $f \sqsubseteq g =_{def} f = g$

lower ordering: $f \leqslant g =_{def} f \leq g$

upper ordering: $f \leqslant g =_{def} f \leq g$

B.3.5 Properties

- df
- st
- sg
- $cs_{\sqsubseteq} \wedge cp_{\sqsubseteq}$
- $cs_{\leqslant} \wedge cp_{\leqslant}$

  Proof: $\leq$ is a total ordering, so $\leqslant$ is both reflexive and transitive. ■

- $cs_{\leqslant} \wedge cp_{\leqslant}$

  Proof: $\leq$ is a total ordering, so $\leqslant$ is both reflexive and transitive. ■

- *reduced.right*
  reduced.bot

  Proof: Let $a \in AS, B \subseteq AS$, s.t. $\{a\} \not\sqsubseteq^* B$.
  Then $\forall b \in B : a \not\sqsubseteq b \Longrightarrow \forall b \in B : a \neq b$
  $\Longrightarrow \forall b \in B : a \not\models b \Longrightarrow a \notin \bigcup_{b \in B} M(b)$
  $\Longrightarrow M(a) \not\subseteq \bigcup_{b \in B} M(b)$. ■

B.4 Card/0

Formulas expressing cardinals, i.e. "at least" properties. So formulas and interpretations are natural numbers, and both the satisfaction and subsumption relations are the decreasing ordering on natural numbers.

B.4.1 Parameters

No parameters.

B.4.2 Syntax

$AS =_{def} \mathbb{N}$

B.4.3 Semantics

domain: $I =_{def} \mathbb{N}$

satisfaction: $i \models f =_{def} i \geq f$

B.4.4 Operations

subsumption: $f \sqsubseteq g =_{def} f \geq g$

top: $\top =_{def} 0$

conjunction: $f \sqcap g =_{def} max(f, g)$

disjunction: $f \sqcap g =_{def} \{min(f, g)\}$

B.4.5 Properties

- *df*
- *st*

  Proof: $\forall f \in \mathbb{N} : f \geq f$
  $\Longrightarrow \forall f \in \mathbb{N} : \exists i \in \mathbb{N} : i \geq f$
  $\Longrightarrow \forall f \in AS : \exists i \in I : i \models f$
  $\Longrightarrow \forall f \in AS : M(f) \neq \emptyset.$ ■

- $cs_{\sqsubseteq}$

  Proof: $f \sqsubseteq g \Longrightarrow f \geq g$
  $\Longrightarrow \forall i \in \mathbb{N} : i \geq f \Rightarrow i \geq g$
  $\Longrightarrow \forall i \in I : i \models f \Rightarrow i \models g$

$\Longrightarrow M(f) \subseteq M(g)$. ■

- $cp_{\sqsubseteq}$

  Proof: $M(f) \subseteq M(g) \Longrightarrow \forall i \in I : i \models f \Rightarrow i \models g$
  $\Longrightarrow \forall i \in \mathbb{N} : i \geq f \Rightarrow i \geq g$
  $\Longrightarrow f \geq g \Longrightarrow f \sqsubseteq g$. ■

- $cp_{\top}$

  Proof: $M(\top) = M(0) = \{i \in \mathbb{N} \mid i \geq 0\} = \mathbb{N} = I$. ■

- $cs_{\sqcap} \wedge cp_{\sqcap}$

  Proof: $M(f \sqcap g) = M(max(f,g))$
  $= \{i \in \mathbb{N} \mid i \geq max(f,g)\}$
  $= \{i \in \mathbb{N} \mid i \geq f \wedge i \geq g\}$
  $= \{i \in \mathbb{N} \mid i \geq f\} \cap \{i \in \mathbb{N} \mid i \geq g\}$
  $= M(f) \cap M(g)$. ■

- $cs_{\sqcap} \wedge cp_{\sqcap}$

  Proof: $M(f \sqcup g) = \bigcup_{h \in f \sqcup g} M(h) = M(max(f,g))$
  $= \{i \in \mathbb{N} \mid i \geq min(f,g)\}$
  $= \{i \in \mathbb{N} \mid i \geq f \vee i \geq g\}$
  $= \{i \in \mathbb{N} \mid i \geq f\} \cup \{i \in \mathbb{N} \mid i \geq g\}$
  $= M(f) \cup M(g)$. ■

- *reduced*

  Proof: Let $A, B \subseteq AS$, s.t. $A \not\sqsubseteq^* B$, $closed_{\sqcap}(A)$, $closed_{\sqcup}(B)$. Then we have

  1. $\forall a_1 \neq a_2 \in A : \neg def(a_1 \sqcap a_2)$: hence, $A = \{a\}$ is a singleton because $\sqcap$ is totally defined.
  2. $\forall b_1 \neq b_2 \in B : b_1 \sqcup b_2 \subseteq B$:
     $\Longrightarrow \forall b_1 \neq b_2 \in B : min(b_1, b_2) \in B$
     $\Longrightarrow min(B) \in B$.
     Now, for all $b \in B$, $b \geq min(B) \Longrightarrow b \sqsubseteq min(B) \Longrightarrow M(b) \subseteq M(min(B))$ (because $cs_{\sqsubseteq}$). Hence $\bigcup_{b \in B} M(b) = M(min(B))$.
  3. $a \not\sqsubseteq min(B) \Longrightarrow M(a) \not\subseteq M(min(B))$ $(cp_{\sqsubseteq})$
     $\Longrightarrow \bigcap_{a \in A} M(a) \not\subseteq \bigcup_{b \in B} M(b)$. ■

B.5 String/0

Formulas are strings and substrings over an arbitrary alphabet plus 2 special characters for representing the beginning and the end of a string. Interpretations are closed strings, i.e., strings having a beginning and an end. Both satisfaction and subsumption relations are based on exact string matching.

B.5.1 Parameters

Σ**:** an alphabet.

B.5.2 Syntax

$AS =_{def} <?\Sigma^* >?$: formulas are strings and substrings

B.5.3 Semantics

domain: $I =_{def} < \Sigma^* >$: interpretations are also strings, but closed with a beginning and an end

satisfaction: $i \models f =_{def} \exists u, v \in AS : i = u.f.v$: f is a substring of i

B.5.4 Operations

subsumption: $f \sqsubseteq g =_{def} \exists u, v \in AS : f = u.g.v$

tautology: $\top =_{def} \epsilon$ (the empty string)

B.5.5 Properties

- *df*
- *st*

  Proof: For every $f \in AS$, there exists a $w \in \Sigma^*$ s.t. $f \in \{< w >, < w, w >, w\}$. Then $< w > \in I$, and $< w > \models f$ in all cases. ■

- $cs_{\sqsubseteq}$

  Proof: $f \sqsubseteq g \Longrightarrow \exists u, v \in AS : f = u.g.v$
  $\Longrightarrow \forall i \in I : i \models f \Rightarrow \exists u', v' \in AS : i = u'.f.v' = u'.u.g.v.v'$
  $\Longrightarrow \forall i \in I : i \models f \Rightarrow i \models g$
  $\Longrightarrow M(f) \subseteq M(g)$. ■

- $cp_{\sqsubseteq}$

  Proof: $f \not\sqsubseteq g \Longrightarrow \nexists u, v \in AS : f = u.g.v$. Then

  - either $f \in < \Sigma^* >$: $f \in I$
    $\Longrightarrow f \models f \wedge f \not\models g$
    $\Longrightarrow M(f) \not\subseteq M(g)$,
  - or $f \in \Sigma^*$: $< f > \in I$, and $< f > \models f$.
    Now suppose $< f > \models g$
    $\Longrightarrow \exists u', v' \in AS :< f >= u'.g.v'$
    $\Longrightarrow \exists u'', v'' \in AS : f = u''.g.v''$
    $\Longrightarrow f \sqsubseteq g \longrightarrow$ contadiction.
    Hence $< f > \not\models g$, and $M(f) \not\subseteq M(g)$.
  - or $f \in < \Sigma^*$: similar proof,
  - or $f \in \Sigma^* >$: similar proof. ■

- $cp_{\top}$

  Proof: $i \in I \Longrightarrow i = i.\epsilon.\epsilon \Longrightarrow i \models \epsilon$
  $\Longrightarrow M(\epsilon) = I$. ■

- $cs_{\sqcap} \wedge cp_{\sqcap}$

- *reduced.right*
  reduced.bot

  Proof: Let $a \in AS, B \subseteq AS$, s.t. $\{a\} \not\sqsubseteq^* B$.
  Then $\forall b \in B : a \not\sqsubseteq b$. Now we have seen in the proof of $cp_{\sqsubseteq}$ that

  $$f \not\sqsubseteq g \Longrightarrow \exists i_f \in I : i \models f \wedge i \not\models g,$$

  where the model i_f depends solely on f (it is the "closure" of f).
  So $\exists i_a \in I : \forall b \in B : i_a \models a \wedge i_a \not\models b$
  $\Longrightarrow \exists i_a \in M(a) : \forall b \in B : i_a \notin M(b)$
  $\Longrightarrow M(a) \not\subseteq \bigcup_{b \in B} M(b)$. ■

Appendix C

Constructors

The Constructors bring structure in formulas and interpretations. They allow for the composition of complex syntaxes and semantics in logics. Recursive structures are often introduced by a recursive composition of these functors. They often mimick data structures in programming languages. Many of these structures can be defined as combinators of simpler functors, hence a great saving of means.

C.1 Sum/2

The disjoint union of syntaxes and interpretation domains. This functor corresponds to sum types in programming languages.

C.1.1 Parameters

$L_1, L_2 \in \mathbb{L}$: 2 logics.

C.1.2 Syntax

$AS =_{def} AS_1 \uplus AS_2 \uplus \{0\}$
$TELL =_{def} TELL_1 \uplus TELL_2$
$ASK =_{def} ASK_1 \uplus ASK_2$

C.1.3 Semantics

domain: $I =_{def} I_1 \uplus I_2$.

satisfaction:

$$M(f) =_{def} \begin{cases} M_1(f) & \text{if } f \in AS_1 \\ M_2(f) & \text{if } f \in AS_2 \\ \emptyset & \text{if } f = 0 \end{cases}$$

C.1.4 Operations

subsumption:

$$f \sqsubseteq g =_{def} \begin{cases} f \sqsubseteq_1 g & \text{if } f, g \in AS_1 \\ f \sqsubseteq_2 g & \text{if } f, g \in AS_2 \\ f \sqsubseteq_1 \bot_1 & \text{if } f \in AS_1 \\ f \sqsubseteq_2 \bot_2 & \text{if } f \in AS_2 \\ f = 0 & \text{otherwise} \end{cases}$$

contradiction: $\bot =_{def} 0$.

conjunction:

$$f \sqcap g =_{def} \begin{cases} f \sqcap_1 g & \text{if } f, g \in AS_1 \\ f \sqcap_2 g & \text{if } f, g \in AS_2 \\ 0 & \text{otherwise} \end{cases}$$

disjunction:

$$f \sqcup g =_{def} \begin{cases} f \sqcup_1 g & \text{if } f, g \in AS_1 \\ f \sqcup_2 g & \text{if } f, g \in AS_2 \\ f \sqcup_d g & \text{otherwise} \end{cases}$$

C.1.5 Properties

- $df \Leftarrow df_1 \wedge df_2$
- $st' \Leftarrow st'_1 \wedge st'_2$

  Proof: $d \in TELL \Longrightarrow d \in TELL_1 \vee d \in TELL_2$
  $\Longrightarrow M_1(d) \neq \emptyset \vee M_2(d) \neq \emptyset$ (st'_1, st'_2)
  $\Longrightarrow M(d) \neq \emptyset$. ■

- $sg' \Leftarrow sg'_1 \wedge sg'_2$

  Proof: Let $d \in TELL$:

  - either $d \in TELL_1 \Longrightarrow Card(M_1(d)) = 1$ (sg'_1)
    $\Longrightarrow Card(M(d)) = 1$.
  - or $d \in TELL_2$: idem.

  ■

- $cs_{\sqsubseteq} \Leftarrow cs_{\sqsubseteq_1} \wedge cs_{\perp_1} \wedge cs_{\sqsubseteq_2} \wedge cs_{\perp_2}$

  Proof: If $f \sqsubseteq g$, then one of the following is true:

  - $f, g \in AS_1$ and $f \sqsubseteq_1 g$
    $\Longrightarrow M_1(f) \subseteq M_1(g)$ $(cs_{\sqsubseteq_1})$
    $\Longrightarrow M(f) \subseteq M(g)$.
  - $f, g \in AS_2$ and $f \sqsubseteq_2 g$
    $\Longrightarrow M_2(f) \subseteq M_2(g)$ $(cs_{\sqsubseteq_2})$
    $\Longrightarrow M(f) \subseteq M(g)$.
  - $f \in AS_1$ and $f \sqsubseteq_1 \perp_1$
    $\Longrightarrow M_1(f) \subseteq \emptyset$ $(cs_{\sqsubseteq_1}, cs_{\perp_1})$
    $\Longrightarrow M(f) = \emptyset \subseteq M(g)$.
  - $f \in AS_2$ and $f \sqsubseteq_2 \perp_2$
    $\Longrightarrow M_2(f) \subseteq \emptyset$ $(cs_{\sqsubseteq_2}, cs_{\perp_2})$
    $\Longrightarrow M(f) = \emptyset \subseteq M(g)$.
  - $f = 0$
    $\Longrightarrow M(f) = \emptyset \subseteq M(g)$.

  Hence in all cases $M(f) \subseteq M(g)$. ■

- $cp_{\sqsubseteq} \Leftarrow cp_{\sqsubseteq_1} \wedge reduced.bot_1 \wedge cp_{\sqsubseteq_2} \wedge reduced.bot_2$

  Proof: If $M(f) \subseteq M(g)$, then one of the following case is true:

- $f, g \in AS_1$
  $\Longrightarrow M_1(f) \subseteq M_1(g) \Longrightarrow f \sqsubseteq_1 g$ $(cp_{\sqsubseteq_1})$
  $\Longrightarrow f \sqsubseteq g$.
- $f, g \in AS_2$, idem as previous case.
- $f \in AS_1$
  $\Longrightarrow M(f) = \emptyset \Longrightarrow M_1(f) = \emptyset \Longrightarrow \{f\} \sqsubseteq_1^* \emptyset$ $(reduced.bot_1)$
  $\Longrightarrow f \sqsubseteq_1 \bot_1$
  $\Longrightarrow f \sqsubseteq g$.
- $f \in AS_2$, idem as previous case.
- $f = 0$
  $\Longrightarrow f \sqsubseteq g$.

Hence, in all cases $f \sqsubseteq g$. ■

- $cp'_{\sqsubseteq} \Leftarrow reduced.bot'_1 \wedge cp'_{\sqsubseteq_1} \wedge reduced.bot'_2 \wedge cp'_{\sqsubseteq_2}$

  Proof: If $d \in TELL, x \in TELL$ and $M(d) \subseteq M(x)$, then one of the following case is true:

  - $d \in TELL_1, x \in ASK_1$
    $\Longrightarrow M_1(d) \subseteq M_1(x) \Longrightarrow d \sqsubseteq_1 x$ $(cp'_{\sqsubseteq_1})$
    $\Longrightarrow d \sqsubseteq x$.
  - $d \in TELL_2, x \in ASK_2$, idem as previous case.
  - $d \in TELL_1$
    $\Longrightarrow M(d) = \emptyset \Longrightarrow M_1(d) = \emptyset$
    $\Longrightarrow \{d\} \sqsubseteq_1^* \emptyset$ $(reduced.bot'_1)$
    $\Longrightarrow d \sqsubseteq_1 \bot_1 \Longrightarrow d \sqsubseteq x$.
  - $d \in TELL_2$, idem as previous case.

  Hence, in all cases $d \sqsubseteq x$. ■

- $cs_\bot$

  Proof: $M(\bot) = M(0) = \emptyset$. ■

- $defst_\sqcap \Leftarrow defst_{\sqcap_1} \wedge defst_{\sqcap_2}$

  Proof: Let $f, g \in AS$, s.t. $M(f) \sqcap M(g) \neq \emptyset$. Then

  - either $f, g \in AS_1$: $M_1(f) \sqcap M_1(g) \neq \emptyset$
    $\Longrightarrow def(f \sqcap_1 g)$ $(defst_{\sqcap_1})$
    $\Longrightarrow def(f \sqcap g)$,

– or $f, g \in AS_2$: idem.

Hence, in all cases, $def(f \sqcap g)$. ■

- $cs_{\sqcap} \Leftarrow cs_{\sqcap_1} \wedge cs_{\sqcap_2}$

  Proof: $f \sqcap g$ is defined in the following cases:

  – $f, g \in AS_1$
  $M_1(f \sqcap_1 g) \subseteq M_1(f) \cap M_1(g)$ $(cs_{\sqcap_1})$
  $\Longrightarrow M(f \sqcap g) \subseteq M(f) \cap M(g)$.
  – $f, g \in AS_2$, idem as previous case.
  – otherwise $M(f \sqcap g) = M(0) = \emptyset \subseteq M(f) \cap M(g)$.

  Hence in all cases, $M(f \sqcap g) \subseteq M(f) \cap M(g)$. ■

- $cp_{\sqcap} \Leftarrow cp_{\sqcap_1} \wedge cp_{\sqcap_2}$

  Proof: $f \sqcap g$ is defined in the following cases:

  – $f, g \in AS_1$
  $M_1(f \sqcap_1 g) \supseteq M_1(f) \cap M_1(g)$ $(cs_{\sqcap_1})$
  $\Longrightarrow M(f \sqcap g) \supseteq M(f) \cap M(g)$.
  – $f, g \in AS_2$, idem as previous case.
  – otherwise $M(f \sqcap g) = M(0) = \emptyset = I_1 \cap I_2 \supseteq M(f) \cap M(g)$.

  Hence in all cases, $M(f \sqcap g) \supseteq M(f) \cap M(g)$. ■

- $cs_{\sqcup} \Leftarrow cs_{\sqcup_1} \wedge cs_{\sqcup_2}$

  Proof: $f \sqcup g$ is defined in the following cases:

  – $f, g \in AS_1$
  $\bigcup_{h \in f \sqcup_1 g} M_1(h) \subseteq M_1(f) \cup M_1(g)$ $(cs_{\sqcup_1})$
  $\Longrightarrow \bigcup_{h \in f \sqcup g} M(h) \subseteq M(f) \cup M(g)$.
  – $f, g \in AS_1$, idem as previous case.
  – otherwise, the default $(cs_{\sqcup_d})$.

  Hence in all cases $\bigcup_{h \in f \sqcup g} M(h) \subseteq M(f) \cup M(g)$. ■

- $cp_{\sqcup} \Leftarrow cp_{\sqcup_1} \wedge cp_{\sqcup_2}$

  Proof: $f \sqcup g$ is defined in the following cases:

- $f, g \in AS_1$
  $\bigcup_{h \in f \sqcup_1 g} M_1(h) \supseteq M_1(f) \cup M_1(g)$ $(cp_{\sqcup_1})$
  $\Longrightarrow \bigcup_{h \in f \sqcup g} M(h) \supseteq M(f) \cup M(g)$.
- $f, g \in AS_1$, idem as previous case.
- otherwise, the default case $(cp_{\sqcup_d})$.

Hence in all cases $\bigcup_{h \in f \sqcup g} M(h) \supseteq M(f) \cup M(g)$. ■

- $reduced(F_1 \cup F_2, G_1 \cup G_2) \Leftarrow reduced_1(F_1, G_1) \wedge reduced_2(F_2, G_2)$
  $reduced.top \Leftarrow reduced.top_1 \wedge reduced.top_2$
  $reduced.bot \Leftarrow reduced.bot_1 \wedge reduced.bot_2$
  $reduced.bot' \Leftarrow reduced.bot'_1 \wedge reduced.bot'_2$
  $reduced.right \Leftarrow reduced.right_1 \wedge reduced.right_2$

  Proof: $F_1 \cup F_2 \not\sqsubseteq^* G_1 \cup G_2 \Longrightarrow F_1 \not\sqsubseteq_1^* G_1 \wedge F_2 \not\sqsubseteq_2^* G_2$
  $\Longrightarrow \bigcap_{f_1 \in F_1} M_1(f_1) \not\subseteq \bigcup_{g_1 \in G_1} M_1(g_1)$ $(reduced_1(F_1, G_1))$
  $\wedge \bigcap_{f_2 \in F_2} M_2(f_2) \not\subseteq \bigcup_{g_2 \in G_2} M_2(g_2)$ $(reduced_2(F_2, G_2))$.
  If $F_2 = \emptyset$, then $\bigcap_{f \in F_1 \cup F_2} M(f) = \bigcap_{f_1 \in F_1} M_1(f_1) \not\subseteq \bigcup_{g_1 \in G_1} M_1(g_1)$
  $\Longrightarrow \bigcap_{f \in F_1 \cup F_2} M(f) \not\subseteq \bigcup_{g \in G_1 \cup G_2} M(g) = \bigcup_{g_1 \in G_1} M_1(g_1) \cup \bigcup_{g_2 \in G_2} M_2(g_2)$
  because $I_1 \cap I_2 = \emptyset$.

  Else if $F_1 = \emptyset$, we get a similar result.

  Otherwise, for all $f_1 \in F_1, f_2 \in F_2$, the conjunction $f_1 \sqcap f_2 = 0$ is defined, which contradicts $F_1 \cup F_2 \not\sqsubseteq^* G_1 \cup G_2$. ■

C.2 Prod/2

The product of syntaxes and interpretation domains. Formulas are couples of sub-formulas, and interpretations are couples of sub-interpretations. This functor corresponds to couples in programming languages, and enables the combination of orthogonal properties such as attributes and values to form valued attributes.

C.2.1 Parameters

$L_1, L_2 \in \mathbb{L}$**:** 2 logics.

C.2.2 Syntax

$AS =_{def} (AS_1 \times AS_2) \uplus \{0\}$
$TELL =_{def} TELL_1 \times TELL_2$
$ASK =_{def} ASK_1 \times ASK_2$

C.2.3 Semantics

domain: $I = I_1 \times I_2$.

satisfaction: $(i_1, i_2) \models f =_{def} f = (f_1, f_2) \wedge i_1 \models_1 f_1 \wedge i_2 \models f_2$.

C.2.4 Operations

subsumption:

$$f \sqsubseteq g =_{def} \bigvee \begin{cases} f = 0 \\ f = (f_1, f_2) \wedge (f_1 \sqsubseteq_1 \bot_1 \vee f_2 \sqsubseteq_2 \bot_2) \\ f = (f_1, f_2) \wedge g = (g_1, g_2) \wedge f_1 \sqsubseteq_1 g_1 \wedge f_2 \sqsubseteq_2 g_2. \end{cases}$$

tautology: $\top =_{def} (\top_1, \top_2)$.

contradiction: $\bot =_{def} 0$.

conjunction:

$$f \sqcap g =_{def} \begin{cases} 0 & \text{if } f = 0 \vee g = 0 \\ (f_1 \sqcap_1 g_1, f_2 \sqcap_2 g_2) & \text{if } f = (f_1, f_2) \wedge g = (g_1, g_2) \end{cases}$$

disjunction:

$$f \sqcup g =_{def} \begin{cases} \{f\} & \text{if } g = 0 \\ \{g\} & \text{if } f = 0 \\ \{(f_1, f_2), (g_1, g_2)\} & \text{if } f = (f_1, f_2) \wedge g = (g_1, g_2) \\ \cup\{(f_1 \sqcap_1 g_1, h_2) \mid def(f_1 \sqcap_1 g_1), h_2 \in f_2 \sqcup_2 g_2\} & \\ \cup\{(h_1, f_2 \sqcap_2 g_2) \mid def(f_2 \sqcap_2 g_2), h_1 \in f_1 \sqcup_1 g_1\} & \end{cases}$$

C.2.5 Properties

- $df \Leftarrow \forall i \in \{1,2\} : df_i$
- $st' \Leftarrow \forall i \in \{1,2\} : st'_i$

  Proof: $\forall i \in \{1,2\} : st'_i \Longrightarrow \forall i \in \{1,2\} : \forall d_i \in TELL_i : M_i(d_i) \neq \emptyset$
  $\Longrightarrow \forall (d_1,d_2) \in TELL : M((d_1,d_2)) = M_1(d_1) \times M_2(d_2) \neq \emptyset$
  $\Longrightarrow st'.$ ■

- $sg' \Leftarrow \forall i \in \{1,2\} : sg'_i$

  Proof: $d \in TELL \Longrightarrow d = (d_1,d_2) \wedge d_1 \in TELL_1 \wedge d_2 \in TELL_2$
  $\Longrightarrow Card(M_1(d_1)) = 1 \wedge Card(M_2(d_2)) = 1$ (sg'_1, sg'_2)
  $\Longrightarrow Card(M_1(d_1) \times M_2(d_2)) = 1$
  $\Longrightarrow Card(M(d)) = 1.$ ■

- $cs_\sqsubseteq \Leftarrow \forall i \in \{1,2\} : cs_{\sqsubseteq_i} \wedge cs_{\bot_i}$

  Proof: $f \sqsubseteq g$ implies

  - either $f = 0$:
    $\Longrightarrow M(f) = \emptyset \Longrightarrow M(f) \subseteq M(g)$.
  - or $f = (f_1,f_2) \wedge f_1 \sqsubseteq_1 \bot_1$
    $\Longrightarrow M_1(f_1) \subseteq M_1(\bot_1)$ $(cs_{\sqsubseteq_1})$
    $\Longrightarrow M_1(f_1) \subseteq \emptyset$ $(cs_{\bot_1})$
    $\Longrightarrow M_1(f_1) \times M_2(f_2) = \emptyset$
    $\Longrightarrow M((f_1,f_2)) = \emptyset$
    $\Longrightarrow M((f_1,f_2)) \subseteq M((g_1,g_2)$
    $\Longrightarrow M(f) \subseteq M(g)$.
  - or $f = (f_1,f_2) \wedge f_2 \sqsubseteq_2 \bot_2$: idem.
  - or $f = (f_1,f_2) \wedge g = (g_1,g_2) \wedge f_1 \sqsubseteq_1 g_1 \wedge f_2 \sqsubseteq_2 g_2$:
    $\forall i \in \{1,2\} : cs_{\sqsubseteq_i}(f_i,g_i)$
    $\Longrightarrow \forall i \in \{1,2\} : f_i \sqsubseteq_i g_i \Rightarrow M_i(f_i) \subseteq M_i(g_i)$
    $\Longrightarrow \forall i \in \{1,2\} : f_i \sqsubseteq_i g_i \Rightarrow M_1(f_1) \times M_2(f_2) \subseteq M_1(g_1) \times M_2(g_2)$
    $\Longrightarrow (f_1,f_2) \sqsubseteq (g_1,g_2) \Rightarrow M((f_1,f_2)) \subseteq M((g_1,g_2))$
    $\Longrightarrow M(f) \subseteq M(g)$.

  Hence in all cases $M(f) \subseteq M(g)$. ■

- $cp_\sqsubseteq(f,g) \Leftarrow \forall i \in \{1,2\} : cp_{\sqsubseteq_i}(f_i,g_i) \wedge reduced.bot_i$
  $cp_\sqsubseteq \Leftarrow \forall i \in \{1,2\} : cp_{\sqsubseteq_i} \wedge reduced.bot_i$
  $cp'_\sqsubseteq \Leftarrow \forall i \in \{1,2\} : cp'_{\sqsubseteq_i} \wedge reduced.bot'_i$

  Proof: $M(f) \subseteq M(g)$ implies

- either $M(f) = \emptyset$:
  $\Longrightarrow f = 0 \vee (f = (f_1, f_2) \wedge (M_1(f_1) = \emptyset \vee M_2(f_2) = \emptyset))$
  $\Longrightarrow f = 0 \vee (f = (f_1, f_2) \wedge (\{f_1\} \sqsubseteq_1^* \emptyset \vee \{f_2\} \sqsubseteq_2^* \emptyset))$ $(reduced.bot_1, reduced.bot_2)$
  $\Longrightarrow f = 0 \vee (f = (f_1, f_2) \wedge f_1 \sqsubseteq_1 \bot_1 \vee f_2 \sqsubseteq_2 \bot_2)$
  $\Longrightarrow f \sqsubseteq g.$
- or $M(f) \neq \emptyset$:
  $\Longrightarrow f = (f_1, f_2) \wedge g = (g_1, g_2) \wedge M((f_1, f_2)) \subseteq M((g_1, g_2))$
  $\Longrightarrow f = (f_1, f_2) \wedge g = (g_1, g_2) \wedge M_1(f_1) \times M_2(f_2) \subseteq M_1(g_1) \times M_2(g_2)$
  $\Longrightarrow f = (f_1, f_2) \wedge g = (g_1, g_2) \wedge M_1(f_1) \subseteq M_1(g_1) \wedge M_2(f_2) \subseteq M_2(g_2)$
  $\Longrightarrow f = (f_1, f_2) \wedge g = (g_1, g_2) \wedge f_1 \sqsubseteq_1 g_1 \wedge f_2 \sqsubseteq g_2 \qquad (\forall i \in \{1, 2\} : cp_{\sqsubseteq_i}(f_i, g_i))$
  $\Longrightarrow f \sqsubseteq g.$

Hence in all cases $f \sqsubseteq g$. ■

- $cp_\top \Leftarrow \forall i \in \{1, 2\} : cp_{\top_i}$

  Proof: $M(\top) = M((\top_1, \top_2)) = M_1(\top_1) \times M_2(\top_2)$
  $= I_1 \times I_2$ $(cp_{\top_1}, cp_{\top_2})$
  $= I.$ ■

- $cs_\bot$

  Proof: $M(\bot) = M(0) = \emptyset.$ ■

- $cs_\sqcup \Leftarrow \forall i \in \{1, 2\} : cs_{\sqcap_i} \wedge cs_{\sqcup_i}$

  Proof:

  - $f = 0$:
    $\Longrightarrow M(f) = \emptyset \Longrightarrow M(g) \subseteq M(f) \cup M(g)$
    $\Longrightarrow \bigcup_{h \in f \sqcup g} M(h) \subseteq M(f) \cup M(g).$
  - $g = 0$: idem.
  - $f = (f_1, f_2), g = (g_1, g_2), h \in f \sqcup g$:
    * $h = f$: $M(f) \subseteq M(f) \cup M(g)$
    * $h = g$: $M(g) \subseteq M(f) \cup M(g)$
    * $h = (f_1 \sqcap_1 g_1, h_2)$, where $def(f_1 \sqcap_1 g_1), h_2 \in f_2 \sqcup_2 g_2$:
      (1) $\bigcup_{h_2 \in f_2 \sqcup_2 g_2} M_2(h_2) \subseteq M_2(f_2) \cup M_2(g_2)$ $(cs_{\sqcup_2})$
      $\Longrightarrow \forall h_2 \in f_2 \sqcup_2 g_2 : M_2(h_2) \subseteq M_2(f_2) \cup M_2(g_2)$
      (2) $M_1(f_1 \sqcap_1 g_1) \subseteq M_1(f_1) \cap M_1(g_1)$ $(cs_{\sqcap_1})$
      Now, $M(h) = M((f_1 \sqcap_1 g_1, h_2)) = M_1(f_1 \sqcap_1 g_1) \times M_2(h_2)$
      $\Longrightarrow M(h) = M_1(f_1 \sqcap_1 g_1) \times ((M_2(h_2) \cap M_2(f_2)) \cup (M_2(h_2) \cap M_2(g_2)))$ (1)
      $\Longrightarrow M(h) \subseteq M_1(f_1 \sqcap_1 g_1) \times M_2(f_2) \cup M_1(f_1 \sqcap_1 g_1) \times M_2(g_2)$

$\Longrightarrow M(h) \subseteq M_1(f_1) \times M_2(f_2) \cup M_1(g_1) \times M_2(g_2)$ (2)
$\Longrightarrow M(h) \subseteq M(f) \cup M(g)$

$*$ $h = (h_1, f_2 \sqcap_2 g_2)$, where $def(f_2 \sqcap_2 g_2), h_1 \in f_1 \sqcup_1 g_1$: idem

So, $\forall h \in f \sqcup g : M(h) \subseteq M(f) \cup M(g)$
$\Longrightarrow \bigcup_{h \in f \sqcup g} M(h) \subseteq M(f) \cup M(g)$. ■

- $cp_{\sqcup}$

  Proof:

  - $f = 0$:
    $\Longrightarrow M(f) = \emptyset \Longrightarrow M(g) \supseteq M(f) \cup M(g)$
    $\Longrightarrow \bigcup_{h \in f \sqcup g} M(h) \supseteq M(f) \cup M(g)$.
  - $g = 0$: idem.
  - $f = (f_1, f_2), g = (g_1, g_2)$:
    $\{f, g\} \subseteq f \sqcup g \Longrightarrow \bigcup_{h \in f \sqcup g} M(h) \supseteq M(f) \cup M(g)$. ■

- $defst_{\sqcap} \Leftarrow \forall i \in \{1, 2\} : defst_{\sqcap_i}$

  Proof: $M(f) \cap M(g) \neq \emptyset$
  $\Longrightarrow M(f) \neq \emptyset \wedge M(g) \neq \emptyset$
  $\Longrightarrow f = (f_1, f_2) \wedge g = (g_1, g_2)$
  $\Longrightarrow (M_1(f_1) \times M_2(f_2)) \cap (M_1(g_1) \times M_2(g_2)) \neq \emptyset$
  $\Longrightarrow (M_1(f_1) \cap M_1(g_1)) \times (M_2(f_2) \cap M_2(g_2)) \neq \emptyset$
  $\Longrightarrow M_1(f_1) \cap M_1(g_1) \neq \emptyset \wedge M_2(f_2) \cap M_2(g_2) \neq \emptyset$
  $\Longrightarrow def(f_1 \sqcap_1 g_1) \wedge def(f_2 \sqcap_2 g_2)$ $(defst_{\sqcap_1}, defst_{\sqcap_2})$
  $\Longrightarrow def(f \sqcap g)$. ■

- $cs_{\sqcap} \Leftarrow \forall i \in \{1, 2\} : cs_{\sqcap_i}$

  Proof:

  - $f = 0$:
    $M(f \sqcap g) = M(0) = \emptyset \subseteq M(f) \cap M(g)$.
  - $g = 0$: idem.
  - $f = (f_1, f_2) \wedge g = (g_1, g_2)$:
    $M((f_1, f_2) \sqcap (g_1, g_2)) = M((f_1 \sqcap_1 g_1, f_2 \sqcap_2 g_2))$
    $= M_1(f_1 \sqcap_1 g_1) \times M_2(f_2 \sqcap_2 g_2)$
    $\subseteq (M_1(f_1) \cap M_1(g_1)) \times (M_2(f_2) \cap M_2(g_2))$ $(cs_{\sqcap_1}, cs_{\sqcap_2})$
    $\subseteq (M_1(f_1) \times M_2(f_2)) \cap (M_1(g_1) \times M_2(g_2))$
    $\subseteq M(f) \cap M(g)$. ■

- $cp_{\sqcap} \Leftarrow \forall i \in \{1,2\} : cp_{\sqcap_i}$

  Proof:

  - $f = 0$:
    $M(f \sqcap g) = M(0) = \emptyset \supseteq M(f) \cap M(g)$.
  - $g = 0$: idem.
  - $f = (f_1, f_2) \wedge g = (g_1, g_2)$:
    $M((f_1, f_2) \sqcap (g_1, g_2)) = M((f_1 \sqcap_1 g_1, f_2 \sqcap_2 g_2))$
    $= M_1(f_1 \sqcap_1 g_1) \times M_2(f_2 \sqcap_2 g_2)$
    $\supseteq (M_1(f_1) \cap M_1(g_1)) \times (M_2(f_2) \cap M_2(g_2))$ $(cp_{\sqcap_1}, cp_{\sqcap_2})$
    $\supseteq (M_1(f_1) \times M_2(f_2)) \cap (M_1(g_1) \times M_2(g_2))$
    $\supseteq M(f) \cap M(g)$. ■

- $reduced.bot \Leftarrow \forall i \in \{1,2\} : reduced.bot_i$
  $reduced.bot' \Leftarrow \forall i \in \{1,2\} : reduced.bot'_i$

  Proof: Let $f \in AS : M(f) \subseteq \emptyset$. Then

  - either $f = 0$: $f \sqsubseteq \bot \Longrightarrow \{f\} \sqsubseteq^* \emptyset$,
  - or $f = (f_1, f_2)$: $M_1(f_1) \times M_2(f_2) = \emptyset$, which implies
    - either $M_1(f_1) = \emptyset$: so $\{f_1\} \sqsubseteq_1^* \emptyset$ $(reduced.bot_i)$
      $\Longrightarrow f_1 \sqsubseteq_1 \bot_1 \Longrightarrow f \sqsubseteq \bot$
      $\Longrightarrow \{f\} \sqsubseteq^* \emptyset$,
    - or $M_2(f_2) = \emptyset$: idem.

  Hence in all cases $\{f\} \sqsubseteq^* \emptyset$. ■

- $reduced.right \Leftarrow \forall i \in \{1,2\} : cs_{\sqsubseteq_i} \wedge cp_{\sqsubseteq_i} \wedge cs_{\bot_i} \wedge cp_{\top_i} \wedge defst_{\sqcap_i} \wedge cp_{\sqcap_i} \wedge reduced.right_i$

  Proof: Let $f \in AS, G \subseteq AS$. We prove that given $M(f) \subseteq \bigcup_{g \in G} M(g)$, we obtain that $\{f\} \sqsubseteq^* G$.

  Firstly, assume that for some $i \in \{1,2\}$, we have $M_i(f_i) \subseteq \bigcup_{g_i \in G_i} M_i(g_i)$
  $\Longrightarrow \{f_i\} \sqsubseteq_i^* \bigsqcup_i G_i)$ $(reduced.right_i)$
  $\Longrightarrow$

$$\bigvee \begin{cases} f_i \sqsubseteq_i \bot_i \\ \exists h_i \in \sqcup_i G_i : \top_i \sqsubseteq_i h_i \\ \exists h_i \in \sqcup_i G_i : f_i \sqsubseteq_i h_i \end{cases}$$

  $\Longrightarrow$

$$\bigvee \begin{cases} M_i(f_i) \subseteq \emptyset & (cs_{\bot_i}) \\ \exists h_i \in \sqcup_i G_i : I_i \subseteq M_i(h_i) & (cp_{\top_i}) \\ \exists h_i \in \sqcup_i G_i : M_i(f_i) \subseteq M_i(h_i) & (cs_{\sqsubseteq_i}) \end{cases}$$

$\Longrightarrow \exists h_i \in \sqcup_i G_i : M_i(f_i) \subseteq M_i(h_i)$
$\Longrightarrow M_i(f_i) \subseteq M_i([\sqcup_i G_i])$, where $[E]$ denotes "some element in the set E".
Hence the equation

$$M_i(f_i) \subseteq \bigcup_{g_i \in G_i} M_i(g_i) \Rightarrow M_i(f_i) \subseteq M_i([\sqcup_i G_i]). \tag{C.1}$$

Secondly, let $K = (M_1(f_1), G_1, \models_1)$ be a formal context, where G_1 denotes the projection of G on the first logic.
Let $\mathcal{G} = \{G' \subseteq G \mid \exists c \in Max(\gamma_K(M_1(f_1))) : (g'_1, g'_2) \in G' \Leftrightarrow g'_1 \in int(c)\}$.
Then we obtain:

- $\forall G' \in \mathcal{G} : ext_K(G'_1) \neq \emptyset$, because G'_1 is the intent of some γ-concept
  $\Longrightarrow \forall G' \in \mathcal{G} : M_1(f_1) \cap \bigcap_{g'_1 \in G'_1} M_1(g'_1) \neq \emptyset$
  $\Longrightarrow \bigcap_{g'_1 \in G'_1} M_1(g'_1) \neq \emptyset$
  $\Longrightarrow def(\sqcap_1 G'_1) \wedge M_1(\sqcap_1 G'_1) \supseteq \bigcap_{g'_1 \in G'_1} M_1(g'_1)$, $(defst_{\sqcap_1}, cp_{\sqcap_1})$
- and $M_1(f_1) \subseteq \bigcup_{G' \in \mathcal{G}} ext_K(G'_1)$, because maximum γ-concepts cover all objects
  $\Longrightarrow M_1(f_1) \subseteq \bigcup_{G' \in \mathcal{G}} (\bigcap_{g'_1 \in G'_1} M_1(g'_1))$, which entails with previous result

$$M_1(f_1) \subseteq \bigcup_{G' \in \mathcal{G}} M_1(\sqcap_1 G'_1). \tag{C.2}$$

Now, starting from $M(f) \subseteq \bigcup_{g \in G} M(g)$, there are 2 cases:

- either $M(f) = \emptyset$: there are again 2 cases:
  - $f = 0$:
    in this case $f \sqsubseteq \bot \Longrightarrow \{f\} \sqsubseteq^* G$.
  - $f = (f_1, f_2)$:
    $\Longrightarrow M_1(f_1) \times M_2(f_2) = \emptyset$
    $\Longrightarrow M_1(f_1) = \emptyset \vee M_2(f_2) = \emptyset$
    $\Longrightarrow M_1(f_1) \subseteq \bigcup_{g_1 \in \emptyset} M_1(g_1) \vee M_2(f_2) \subseteq \bigcup_{g_2 \in \emptyset} M_2(g_2)$
    $\Longrightarrow \{f_1\} \sqsubseteq^*_1 \emptyset) \vee \{f_2\} \sqsubseteq^*_2 \emptyset)$ $(reduced.right_1, reduced.right_2)$
    $\Longrightarrow f = (f_1, f_2) \wedge (f_1 \sqsubseteq_1 \bot_1 \vee f_2 \sqsubseteq_2 \bot_2)$
    $\Longrightarrow f \sqsubseteq \bot \Longrightarrow \{f\} \sqsubseteq^* G$.
- or $M(f) \neq \emptyset$:
  $\Longrightarrow f = (f_1, f_2) \wedge M_1(f_1) \times M_2(f_2) \neq \emptyset$
  $\Longrightarrow M_1(f_1) \neq \emptyset \wedge M_2(f_2) \neq \emptyset$.
  Now $M(f) \subseteq \bigcup_{g \in G} M(g)$
  $\Longrightarrow M_1(f_1) \times M_2(f_2) \subseteq \bigcup_{g \in G} (M_1(g_1) \times M_2(g_2))$
  $\Longrightarrow \forall i_1, i_2 : i_1 \in M_1(f_1) \wedge i_2 \in M_2(f_2) \Rightarrow \exists (g_1, g_2) \in G : i_1 \in M_1(g_1) \wedge i_2 \in M_2(g_2)$
  $\Longrightarrow \forall G' \in \mathcal{G} : \forall i_1, i_2 : i_1 \in M_1(f_1) \cap \bigcap_{g'_1 \in G'_1} M_1(g'_1) \wedge i_2 \in M_2(f_2) \Rightarrow \exists (g_1, g_2) \in$

$G : i_1 \in M_1(g_1) \wedge i_2 \in M_2(g_2)$
$\Longrightarrow \forall G' \in \mathcal{G} : \forall i_1, i_2 : i_1 \in \gamma_K^{-1}(\mu_K(G'_1)) \wedge i_2 \in M_2(f_2) \Rightarrow \exists (g'_1, g'_2) \in G' : i_1 \in M_1(g'_1) \wedge i_2 \in M_2(g'_2)$, because every $i_1 \in \gamma_K^{-1}(\mu_K(G'_1))$ belongs only to the extent of the $g_1 \in G_1$ that are also in G'_1
$\Longrightarrow \forall G' \in \mathcal{G} : \forall i_2 : i_2 \in M_2(f_2) \Rightarrow \exists g'_2 \in G'_2 : i_2 \in M_2(g'_2)$
$\Longrightarrow \forall G' \in \mathcal{G} : M_2(f_2) \subseteq \bigcup_{g'_2 \in G'_2} M_2(g'_2)$
$\Longrightarrow \forall G' \in \mathcal{G} : M_2(f_2) \subseteq M_2([\sqcup_2 G'_2])$ (C.1)
$\Longrightarrow M_2(f_2) \subseteq \bigcap_{G' \in \mathcal{G}} M_2([\sqcup_2 G'_2])$
$\Longrightarrow \bigcap_{G' \in \mathcal{G}} M_2([\sqcup_2 G'_2]) \neq \emptyset$, as $M_2(f_2) \neq \emptyset$
$\Longrightarrow x_2 = \sqcap_{G' \in \mathcal{G}} [\sqcup_2 G'_2] \wedge def(x_2) \wedge M_2(x_2) \supseteq \bigcap_{G' \in \mathcal{G}} M_2([\sqcup_2 G'_2])$ $(defst_{\sqcap_2}, cp_{\sqcap_2})$
which implies

$$x_2 = \sqcap_{G' \in \mathcal{G}} [\sqcup_2 G'_2] \wedge M_2(f_2) \subseteq M_2(x_2). \tag{C.3}$$

Also, from (C.2), $M_1(f_1) \subseteq \bigcup_{G' \in \mathcal{G}} M_1(\sqcap_1 G'_1)$, which implies by (C.1)

$$x_1 = [\sqcup_{G' \in \mathcal{G}} \sqcap_1 G'_1] \wedge M_1(f_1) \subseteq M_1(x_1). \tag{C.4}$$

Hence, by putting (C.4) and (C.3) together, we obtain
$M_1(f_1) \times M_2(f_2) \subseteq M_1(x_1) \times M_2(x_2)$
$\Longrightarrow M((f_1, f_2)) \subseteq M((x_1, x_2))$,
where $(x_1, x_2) = ([\sqcup_1 H_1], \sqcap_2 H_2)$,
where $H = \{(\sqcap_1 G'_1, [\sqcup_2 G'_2]) \mid G' \in \mathcal{G}\}$.
From the definition of $\sqcup$, extended as in (C.1), we have $H \subseteq \sqcup G$, and $(x_1, x_2) \in \sqcup H$. Hence $(x_1, x_2) \in \sqcup G$
$\Longrightarrow \exists x \in \sqcup G : M(f) \subseteq M(x)$
$\Longrightarrow \exists x \in \sqcup G : f \sqsubseteq x$ $(cp_{\sqsubseteq})$
$\Longrightarrow \{f\} \sqsubseteq^* G$. ■

C.3 Multiset/1

The multisets of sub-formulas and interpretations. These can also be seen as unordered tuples. Multisets can contain several equivalent elements, so that it can be expressed that a multiset contains "at least n elements of some type". It can also be expressed that 2 properties are satisfied by 2 different elements.

C.3.1 Parameters

$E \in \mathbb{L}$**:** a logic, whose formulas play the role of multiset elements.

C.3.2 Syntax

$AS =_{def} \mathbb{N}^{AS_E}$
$TELL =_{def} \mathbb{N}^{TELL_E}$
$ASK =_{def} \mathbb{N}^{ASK_E}$

Formulas are multisets, whose elements are formulas of E. A possible concrete syntax is $\{e_1, ..., e_n\}$. A shorthand for $e, ..., e$ (n times the same element e) is $n\ e$; and a shorthand for $\{n\ e\}$ is simply $n\ e$. In this way the property "contains at least n elements of type e" can be expressed as $n\ e$. This can be combined for different numbers and types of elements as in $\{n_1\ e_1, n_2\ e_2\}$, whose meaning is "contains n_1 different elements of type e_1, and n_2 other elements of type e_2".

C.3.3 Semantics

domain: $I =_{def} \mathbb{N}^{I_E}$ (multisets of I_E interpretations).

satisfaction: $i \models E =_{def} \exists \rho \in E \hookrightarrow i : \forall e \in E : \rho(e) \models_E e$,
where $\hookrightarrow$ denotes an injective function.

C.3.4 Operations

subsumption: $E \sqsubseteq F =_{def} \exists \rho \in F \hookrightarrow E : \forall f \in F : \rho(f) \sqsubseteq_E f$

tautology: $\top =_{def} \{\}$.

C.3.5 Properties

- $df \Leftarrow df_E$
- $st' \Leftarrow st'_E$

  Proof: $E \in TELL \Longrightarrow \forall e \in E : e \in TELL_E$
  $\Longrightarrow \forall e \in E : \exists i_E \in I_E : i_E \models_E e$

$\Longrightarrow \exists i \in I : \exists \rho \in E \hookrightarrow i : \forall e \in E : \rho(e) \models_E e$
$\Longrightarrow \exists i \in I : i \models E.$ ■

- $cs_{\sqsubseteq}(E,F) \Leftarrow \forall e \in E, f \in F : cs_{\sqsubseteq_E}(e,f)$
  $cs_{\sqsubseteq} \Leftarrow cs_{\sqsubseteq_E}$

  Proof: $E \sqsubseteq F \longrightarrow \exists \rho \in F \hookrightarrow E : \forall f \in F : \rho(f) \sqsubseteq_E f.$
  $i \models E \Longrightarrow \exists \rho' \in E \hookrightarrow i : \forall e \in E : \rho'(e) \models_E e$
  $\Longrightarrow \exists \rho' \in E \hookrightarrow i : \forall f \in F : \rho'(\rho(f)) \models_E \rho(f) \sqsubseteq_E f$ hypothesis
  $\Longrightarrow \exists \rho'' = \rho \circ \rho' \in F \hookrightarrow i : \forall f \in F : \rho'(\rho(f)) \models_E f$ $\forall e \in E, f \in F : cs_{\sqsubseteq_E}(e,f)$
  $\Longrightarrow \exists \rho'' \in F \hookrightarrow i : \forall f \in F : \rho''(f) \models_E f$
  $\Longrightarrow i \models F.$
  Hence $M(E) \subseteq M(F)$. ■

- $cp_{\sqsubseteq}(E,F) \Leftarrow \forall e \in E : sg(e) \wedge \forall e \in E, f \in F : cp_{\sqsubseteq_E}(e,f)$
  $cp'_{\sqsubseteq} \Leftarrow cp'_{\sqsubseteq_E} \wedge sg'_E$

  Proof: We prove that $E \not\sqsubseteq F$ implies $\exists i \in I : i \models E \wedge i \not\models F.$

  Firstly, $E \not\sqsubseteq F$
  $\Longrightarrow \forall \rho \in F \hookrightarrow E : \exists f \in F : \rho(f) \not\sqsubseteq_E f$
  $\Longrightarrow \forall \rho \in F \hookrightarrow E : \exists f \in F : M_E(\rho(f)) \not\subseteq M_E(f).$ $\forall e \in E, f \in F : cp_{\sqsubseteq_E}(e,f)$

  Secondly, let $i = \biguplus_{e \in E} M_E(e)$ multiset union
  and $\forall e \in E : \rho'(e) = choice(M_E(e))$. $(\forall e \in E : sg_E(e))$
  This implies that ρ' is a bijection from E to i
  $\Longrightarrow \exists \rho' \in E \hookrightarrow i : \forall e \in E : \rho'(e) \models_E e$
  $\Longrightarrow i \models E.$

  Now, let $\rho'' \in F \hookrightarrow i$
  $\Longrightarrow \exists \rho \in F \hookrightarrow E : \rho'' = \rho \circ \rho'$ ρ' is a bijection
  $\Longrightarrow \exists \rho \in F \hookrightarrow E : \rho'' = \rho \circ \rho', \exists f \in F : M_E(\rho(f)) \not\subseteq M_E(f)$ hypothesis
  $\Longrightarrow ... \exists f \in F : \rho'(\rho(f)) \notin M_E(f)$ hypothesis
  $\Longrightarrow ... \exists f \in F : \rho''(f) \not\models_E f$
  $\Longrightarrow \exists f \in F : \rho''(f) \not\models_E f.$
  Hence $\forall \rho'' \in F \hookrightarrow i : \exists f \in F : \rho''(f) \not\models_E f$
  $\Longrightarrow i \not\models f.$ ■

- $cp_{\top}$

  Proof: Trivial from definition of $\top$, and semantics. ■

C.4 Pair/1 = $\lambda X.Prod(X, X)$

Formulas and interpretations are ordered pairs over some sub-logic.

C.5 Attrval/1 = $\lambda Val.Prod(Atom, Val)$

The valued attributes, where values are taken in the sub-logic *Val*.

C.6 Option/1 = $\lambda X.Sum(Unit, X)$

This combinator helps to make a property optional by allowing a *Unit*-value to be used instead.

C.7 Vector/1 = $\mu V.\lambda X.Option(Prod(X, V))$

This combinator defines vectors as a generalization of pairs to an arbitrary number of elements (starting from 0). This is close to arrays in programming languages.

C.8 List/1 = $\mu L.\lambda X.Top(Option(Prod(X, L)))$

Lists are similar to vectors, except that in *ASK*-formulas only a prefix of a list can be specified.

C.9 Tree/2 = $\mu T.\lambda F.\lambda X.Top(Prod(X, F(T)))$

This combinator defines trees in a very generic way. X abstracts the labelling of nodes, and F is a functor that abstracts the way the children of a node are organized.

C.10 NaryTree/1 = $\lambda X.Tree(List, X)$

An n-ary tree is a tree in which each node has a list of children (possibly empty).

C.11 BinaryTree/1 = $\lambda X.Tree(\lambda T.Option(Pair(T)), X)$

A binary tree is a tree in which each node may have a pair of children.

Appendix D

Abstractors

The abstractors extend the syntax, especially the set of *ASK*-formulas, while keeping the interpretation domain unchanged. In information systems, this is mainly used to enhance the expressivity of queries, while keeping the description language unchanged. This is usually done by adding more abstract/general formulas (e.g., intervals, boolean connectives).

D.1 Top/1

Addition of a top to the syntax of a logic.

D.1.1 Parameters

$X \in \mathbb{L}$**:** the logic to which a top is to be added.

D.1.2 Syntax

$AS =_{def} AS_X \uplus \{1\}$
$TELL =_{def} TELL_X$
$ASK =_{def} ASK_X \uplus \{1\}$

D.1.3 Semantics

domain: $I = I_X$.

satisfaction:

$$i \models f =_{def} \begin{cases} true & \text{if } f = 1 \\ i \models_X f & \text{if } f \in AS_X \end{cases}$$

D.1.4 Operations

subsumption:

$$f \sqsubseteq g =_{def} \begin{cases} true & \text{if } g = 1 \\ \top_X \sqsubseteq_X g & \text{if } f = 1 \wedge g \in AS_X \\ f \sqsubseteq_X g & \text{if } f, g \in AS_X \end{cases}$$

tautology: $\top =_{def} 1$.

contradiction: $\bot =_{def} \bot_X$.

conjunction:

$$f \sqcap g =_{def} \begin{cases} f & \text{if } g = 1 \\ g & \text{if } f = 1 \\ f \sqcap_X g & \text{if } f, g \in AS_X \end{cases}$$

disjunction:

$$f \sqcup g =_{def} \begin{cases} \{1\} & \text{if } f = 1 \vee g = 1 \\ f \sqcup_X g & \text{if } f, g \in AS_X \end{cases}$$

D.1.5 Properties

- $df \Leftarrow df_X$
- $st' \Leftarrow st'_X$

  Proof: $d \in TELL \Longrightarrow d \in TELL_X$
  $\Longrightarrow M_X(d) \neq \emptyset$ (st'_X)
  $\Longrightarrow M(d) \neq \emptyset$. ■

- $sg' \Leftarrow sg'_X$

  Proof: $d \in TELL \Longrightarrow d \in TELL_X \Longrightarrow Card(M_X(d)) = 1$ (sg'_X)
  $\Longrightarrow Card(M(d)) = 1$. ■

- $cs_\sqsubseteq \Leftarrow cs_{\sqsubseteq X} \wedge cp_{\top X}$

  Proof: $f \sqsubseteq g$ implies

  - either $g = 1$:
    $\Longrightarrow M(g) = I \Longrightarrow M(f) \subseteq M(g)$.
  - or $f = 1, g \in AS_X, \top_X \sqsubseteq_X g$ $(def(\top_X))$:
    $\Longrightarrow M_X(\top_X) \subseteq M_X(g)$ $(cs_{\sqsubseteq X})$
    $\Longrightarrow I_X \subseteq M_X(g)$ $(cp_{\top X})$
    $\Longrightarrow M(f) \subseteq M(g) = I$.
  - or $f, g \in AS_X, f \sqsubseteq_X g$:
    $\Longrightarrow M_X(f) \subseteq M_X(g)$ $(cs_{\sqsubseteq X})$
    $\Longrightarrow M(f) \subseteq M(g)$.

  Hence in all cases $M(f) \subseteq M(g)$. ■

- $cp_\sqsubseteq \Leftarrow cp_{\sqsubseteq X} \wedge reduced.top_X$

  Proof: $M(f) \subseteq M(g)$ implies

  - either $g = 1$:
    $\Longrightarrow f \sqsubseteq g$.
  - or $f = 1, g \in AS_X$:
    $\Longrightarrow M(f) = I \subseteq M(g)$
    $\Longrightarrow \bigcap_{k \in \emptyset} M(k) \subseteq \bigcup_{l \in \{g\}} M(l)$
    $\Longrightarrow \emptyset \sqsubseteq^*_X \{g\}$ $(reduced.top_X)$
    $\Longrightarrow \top_X \sqsubseteq g \Longrightarrow f \sqsubseteq g$.

– or $f, g \in AS_X$:
$\Longrightarrow M_X(f) \subseteq M_X(g)$
$\Longrightarrow f \sqsubseteq_X g$ $(cp_{\sqsubseteq_X})$
$\Longrightarrow f \sqsubseteq g$.

Hence in all cases $f \sqsubseteq g$. ■

- $cp'_{\sqsubseteq} \Leftarrow cp'_{\sqsubseteq_X}$
Proof: Let $d \in TELL, x \in ASK$.
$M(f) \subseteq M(g)$ implies

– either $g = 1$:
$\Longrightarrow d \sqsubseteq x$.

– or $d \in TELL_X, x \in ASK_X$:
$\Longrightarrow M_X(d) \subseteq M_X(x)$
$\Longrightarrow d \sqsubseteq_X x$ $(cp'_{\sqsubseteq_X})$
$\Longrightarrow d \sqsubseteq x$.

Hence in all cases $d \sqsubseteq x$. ■

- $cp_\top$
Proof: $M(\top) = M(1) = I$. ■

- $cs_\bot \Leftarrow cs_{\bot_X}$
Proof: $M(\bot) = M(\bot_X) = M_X(\bot_X) = \emptyset$ $(cs_{\bot_X})$. ■

- $defst_\sqcap \Leftarrow defst_{\sqcap_X}$
Proof: Assume $M(f) \cap M(g) \neq \emptyset$.
If $f = 1$ or $g = 1$, then $f \sqcap g$ is defined. Otherwise, $f, g \in AS_X$
$\Longrightarrow M_X(f) \cap M_X(g) \neq \emptyset$
$\Longrightarrow def(f \sqcap_X g)$ $(defst_{\sqcap_X})$
$\Longrightarrow def(f \sqcap g)$. ■

- $cs_\sqcap \Leftarrow cs_{\sqcap_X}$
- $cp_\sqcap \Leftarrow cp_{\sqcap_X}$
- $cs_\sqcup \Leftarrow cs_{\sqcup_X}$
- $cp_\sqcup \Leftarrow cp_{\sqcup_X}$

- $reduced(F,G) \Leftarrow 1 \in G \vee (1 \notin G \wedge reduced_X(F \setminus \{1\}, G))$
  $reduced \Leftarrow reduced_X$
  $reduced.bot \Leftarrow reduced.bot_X$
  $reduced.bot' \Leftarrow reduced.bot'_X$
  $reduced.top \Leftarrow reduced.top_X$
  $reduced' \Leftarrow reduced'_X$

  Proof: Let us consider 2 cases:

  - either $1 \in G$:
    $\Longrightarrow \exists g \in G : \top \sqsubseteq g$
    $\Longrightarrow F \sqsubseteq^* G$.
  - or $1 \notin G$:
    $\bigcap_{f \in F} M(f) \subseteq \bigcup_{g \in G} M(g)$
    $\Longrightarrow \bigcap_{f \in F \setminus \{1\}} M(f) \subseteq \bigcup_{g \in G} M(g)$
    $\Longrightarrow F \setminus \{1\} \sqsubseteq^*_X G$ ($reduced_X(F \setminus \{1\}, G)$)
    $\Longrightarrow F \sqsubseteq^* G$ (definition of $\sqsubseteq^*$).

  Hence in all cases $\bigcap_{f \in F} M(f) \subseteq \bigcup_{g \in G} M(g) \Rightarrow F \sqsubseteq^* G$. ■

D.2 Interval/1

The intervals over values that are formulas of another logic. This supposes the sub-logic has an ordering over its interpretations, and has its operations $\leqslant$ and $\leqslant$ defined.

D.2.1 Parameters

$V \in \mathbb{L}$**:** a logic of values, equiped with operators $\leqslant$, and $\leqslant$, and whose semantics is equipped with an ordering $\leq$.

D.2.2 Syntax

$AS =_{def} AS_V \times AS_V$
$ASK =_{def} ASK_V \times ASK_V$
$TELL =_{def} \{(d,d) \mid d \in TELL_V\}$

Formula (a,b), where $a,b \in AS_V$, is only well-formed when $a \leqslant_V b$ or $a \leqslant_V b$ (ensures the interval is not empty).

D.2.3 Semantics

domain: $I =_{def} I_V$

ordering: $i \leq j =_{def} i \leq_V j)$

satisfaction: $i \models (a,b) =_{def} \exists i_a \in M_V(a), i_b \in M_V(b) : i_a \leq_V i \leq_V i_b$

D.2.4 Operations

subsumption: $(a_1,b_1) \sqsubseteq (a_2,b_2) =_{def} a_2 \leqslant_V a_1 \wedge b_1 \leqslant_V b_2$

D.2.5 Properties

- $df \Leftarrow df_V$
- $st \Leftarrow st_V \wedge cs_{\leqslant_V} \wedge cs_{\leqslant_V}$
  $st' \Leftarrow st'_V \wedge cs_{\leqslant_V} \wedge cs_{\leqslant_V}$

  Proof: $(a,b) \in AS \Longrightarrow a,b \in AS_V \wedge (a \leqslant_V b \vee a \leqslant_V b)$
  $\Longrightarrow M_V(a) \neq \emptyset \wedge M_V(b) \neq \emptyset.$ (st_V)
  Now,

– either $a \leqslant_V b$:
$\Longrightarrow \forall i_b \in M_V(b) : \exists i_a \in M_V(a) : i_a \leq_V i_b$ $(cs_{\leqslant_V})$
$\Longrightarrow \exists i_b \in M_V(b), i_a \in M_V(a) : i_a \leq_V i_b$ $(M_V(b) \neq \emptyset)$
$\Longrightarrow \exists i \in M((a,b)) \Longrightarrow M((a,b)) \neq \emptyset$.

– or $a \leqslant_V b$:
$\Longrightarrow \forall i_a \in M_V(a) : \exists i_b \in M_V(b) : i_a \leq_V i_b$ $(cs_{\leqslant_V})$
$\Longrightarrow \exists i_a \in M_V(a), i_b \in M_V(b) : i_a \leq_V i_b$ $(M_V(a) \neq \emptyset)$
$\Longrightarrow \exists i \in M((a,b)) \Longrightarrow M((a,b)) \neq \emptyset$. ■

- $sg' \Leftarrow {sg'}_V$

Proof: Let $(d,d) \in TELL$. Suppose some interpretation $i \in I$ s.t. $i \models (d,d)$
$\Longrightarrow \exists i_a, i_b \in M_V(d) : i_a \leq_V i \leq_V i_b$
$\Longrightarrow i_d \leq_V i \leq_V i_d$, where i_d is the unique model of d $({sg'}_V)$
$\Longrightarrow i = i_d$.
Hence $M((d,d)) = M_V(d) = \{i_d\}$. ■

- $cs_{\sqsubseteq} \Leftarrow cs_{\leqslant_V} \wedge cs_{\leqslant_V}$

Proof: $(a_1,b_1) \sqsubseteq (a_2,b_2)$
$\Longrightarrow a_2 \leqslant a_1 \wedge b_1 \leqslant b_2$
$\Longrightarrow \forall i_1 \in M_V(a_1) : \exists i_2 \in M_V(a_2) : i_2 \leq i_1$ $(cs_{\leqslant_V})$
$\wedge \forall i_1 \in M_V(b_1) : \exists i_2 \in M_V(b_2) : i_1 \leq i_2$ $(cs_{\leqslant_V})$

Now $i \models (a_1,b_1)$
$\Longrightarrow \exists i_1 \in M_V(a_1) : i_1 \leq i$
$\wedge \exists i_1 \in M_V(b_1) : i \leq i_1$
$\Longrightarrow \exists i_1 \in M_V(a_1), i_2 \in M_V(a_2) : i_2 \leq i_1 \leq i$
$\wedge \exists i_1 \in M_V(b_1), i_2 \in M_V(b_2) : i \leq i_1 \leq i_2$
$\Longrightarrow \exists i_2 \in M_V(a_2) : i_2 \leq i \wedge \exists i_2 \in M_V(b_2) : i \leq i_2$
$\Longrightarrow i \models (a_2,b_2)$.

Hence $M((a_1,b_1)) \subseteq M((a_2,b_2))$. ■

- $cp_{\sqsubseteq} \Leftarrow cp_{\leqslant_V} \wedge cp_{\leqslant_V}$

Proof: $M((a_1,b_1)) \subseteq M((a_2,b_2))$
$\Longrightarrow \forall i \in I : i \models (a_1,b_1) \Rightarrow i \models (a_2,b_2)$
$\Longrightarrow \forall i \in I : (\exists i_a \in M_V(a_1) : i_b \in M_V(b_1) : i_a \leq i \leq i_b)$
$\Rightarrow (\exists i_a \in M_V(a_2) : i_b \in M_V(b_2) : i_a \leq i \leq i_b)$
$\Longrightarrow \forall i \in M_V(a_1) : (\exists i_a \in M_V(a_1), i_b \in M_V(b_1) : i_a \leq i \leq i_b) \Rightarrow \exists i_2 \in M_V(a_2) : i_2 \leq i$
Then choose $i_a = i = i_b$:

$\Longrightarrow \forall i_1 \in M_V(a_1) : \exists i_2 \in M_V(a_2) : i_2 \leq i_1$
$\Longrightarrow a_2 \leqslant a_1$ $(cp_{\leqslant_V})$.

Similarly, we get the similar result with b_2 $(cp_{\leqslant_V})$:
$\forall i_1 \in M_V(b_1) : \exists i_2 \in M_V(b_2) : i_1 \leq i_2$
$\Longrightarrow b_1 \leqslant b_2$.
These 2 propositions are the definition of $(a_1, b_1) \sqsubseteq (a_2, b_2)$, which terminates the proof. ■

D.3 Bounds/1 $\supseteq \lambda X.Sum(X, Sum(\{-\infty\}, \{+\infty\}))$

Add bounds w.r.t. the ordering over formulas (as used by *Interval*). These bounds are added as both formulas and interpretations. This functor is defined as an extension of the combinator given in the section title ($\{-\infty\}$ stands for the logic functor *Unit* where $-\infty$ is the only formula). This means it inherits everything from this combinator, and possibly redefines part of it.

D.3.1 Parameters

$X \in \mathbb{L}$**:** a logic of values.

D.3.2 Semantics

ordering:

$$i \leq j =_{def} \bigvee \begin{cases} i = -\infty \\ j = +\infty \\ i, j \in I_X \wedge i \leq_X j \end{cases}$$

D.3.3 Operations

lower ordering:

$$f \leqslant g =_{def} \bigvee \begin{cases} g = 0 \\ f = -\infty \\ g = +\infty \\ f, g \in AS_X \wedge f \leqslant_X g \end{cases}$$

upper ordering:

$$f \leqslant g =_{def} \bigvee \begin{cases} f = 0 \\ f = -\infty \\ g = +\infty \\ f, g \in AS_X \wedge f \leqslant_X g \end{cases}$$

D.3.4 Properties

- $cs_{\leqslant} \Leftarrow cs_{\leqslant_X} \wedge st$

  Proof: $f \leqslant g$ implies:

  - either $g = 0$: $M(g) = \emptyset \Longrightarrow \forall j \in M(g) : \exists i \in M(f) : i \leq j$.
  - or $f = -\infty$:
    $\forall j \in M(g) : -\infty \leq j \Longrightarrow \forall j \in M(g) : \exists i \in M(f) : i \leq j$.

- or $g = +\infty$: then,
  - $f = -\infty$: $\forall j \in M(g) : \exists i \in M(f) : i \leq j$ $(i = -\infty, j = +\infty)$
  - $f = +\infty$: $\forall j \in M(g) : \exists i \in M(f) : i \leq j$ $(i = j = +\infty)$
  - $f \in AS_X$: $M(f) = M_X(f) \neq \emptyset$ (st_X)
    $\Longrightarrow \exists i \in M(f) : i \leq +\infty$
    $\Longrightarrow \forall j \in M(g) : \exists i \in M(f) : i \leq j.$
- or $f, g \in AS_X \wedge f \leqslant_X g$:
  $\Longrightarrow \forall j \in M_X(g) : \exists i \in M_X(f) : i \leq_X j$ $(cs_{\leqslant_X})$
  $\Longrightarrow \forall j \in M(g) : \exists i \in M(f) : i \leq j.$ $(Sum).$

Hence in all cases, $\forall j \in M(g) : \exists i \in M(f) : i \leq j.$ ■

- $cp_{\leqslant} \Leftarrow cp_{\leqslant_X} \wedge st_X$

  Proof: $f \not\leqslant g$ implies

  - either $f = 0 \wedge g \neq 0$:
    - $g = -\infty$: $M(g) \neq \emptyset$
    - $g = +\infty$: $M(g) \neq \emptyset$
    - $g \in AS_X$: $M(g) = M_X(g) \neq \emptyset$ (st_X)

    Hence in all cases, $M(g) \neq \emptyset$, and $M(f) = \emptyset$
    $\Longrightarrow \exists j \in M(g) : \forall i \in M(f) : i \not\leq j.$
  - or $f = +\infty \wedge g = -\infty$: $\exists j \in M(g) : \forall i \in M(f) : i \leq j$
  - or $f = +\infty \wedge g \in AS_X$: $M(g) = M_X(g) \neq \emptyset$ (st_X)
    $\Longrightarrow \exists j \in M(g) : +\infty \not\leq j$
    $\Longrightarrow \exists j \in M(g) : \forall i \in M(f) : i \not\leq j.$
  - or $f \in AS_X \wedge g = -\infty$:
    $\Longrightarrow \forall i \in M(f) : i \not\leq -\infty$
    $\Longrightarrow \exists j \in M(g) : \forall i \in M(f) : i \not\leq j.$
  - or $f, g \in AS_X \wedge f \not\leqslant_X g$:
    $\Longrightarrow \exists j \in M_X(g) : \forall i \in M_X(f) : i \not\leq_X j$ $(cp_{\leqslant_X})$
    $\Longrightarrow \exists j \in M(g) : \forall i \in M(f) : i \not\leq j.$ (Sum)

  Hence in all cases, $\neg\forall j \in M(g) : \exists i \in M(f) : i \leq j.$ ■

- $cs_{\leqslant} \Leftarrow cs_{\leqslant_X} \wedge st$

  Proof: $f \leqslant g$ implies:

  - either $f = 0$: $M(f) = \emptyset \Longrightarrow \forall i \in M(f) : \exists j \in M(g) : i \leq j.$

- or $g = +\infty$:
  $\forall i \in M(f) : i \leq +\infty \Longrightarrow \forall i \in M(f) : \exists j \in M(g) : i \leq j$.
- or $f = -\infty$: then,
  - $g = +\infty$: $\forall i \in M(f) : \exists j \in M(g) : i \leq j$ $(i = -\infty, j = +\infty)$
  - $g = -\infty$: $\forall i \in M(f) : \exists j \in M(g) : i \leq j$ $(i = j = -\infty)$
  - $g \in AS_X$: $M(g) = M_X(g) \neq \emptyset$ (st_X)
    $\Longrightarrow \exists j \in M(g) : -\infty \leq j$
    $\Longrightarrow \forall i \in M(f) : \exists j \in M(g) : i \leq j$.
- or $f, g \in AS_X \wedge f \preccurlyeq_X g$:
  $\Longrightarrow \forall i \in M_X(f) : \exists j \in M_X(g) : i \leq_X j$ $(cs_{\preccurlyeq_X})$
  $\Longrightarrow \forall i \in M(f) : \exists j \in M(g) : i \leq j$. (Sum).

Hence in all cases, $\forall i \in M(f) : \exists j \in M(g) : i \leq j$. ■

- $cp_{\preccurlyeq} \Leftarrow cp_{\preccurlyeq_X} \wedge st_X$

  Proof: $f \not\preccurlyeq g$ implies
  - either $f \neq 0 \wedge g = 0$:
    - $f = -\infty$: $M(f) \neq \emptyset$
    - $f = +\infty$: $M(f) \neq \emptyset$
    - $f \in AS_X$: $M(f) = M_X(f) \neq \emptyset$ (st_X)

    Hence in all cases, $M(f) \neq \emptyset$, and $M(g) = \emptyset$
    $\Longrightarrow \exists i \in M(f) : \forall j \in M(g) : i \not\leq j$.
  - or $f = +\infty \wedge g = -\infty$: $\exists i \in M(f) : \forall j \in M(g) : i \leq j$
  - or $f \in AS_X \wedge g = -\infty$: $M(f) = M_X(f) \neq \emptyset$ (st_X)
    $\Longrightarrow \exists i \in M(f) : i \not\leq -\infty$
    $\Longrightarrow \exists i \in M(f) : \forall j \in M(g) : i \not\leq j$.
  - or $f = +\infty \wedge g \in AS_X$:
    $\Longrightarrow \forall j \in M(g) : +\infty \not\leq j$
    $\Longrightarrow \exists i \in M(f) : \forall j \in M(g) : i \not\leq j$.
  - or $f, g \in AS_X \wedge f \not\preccurlyeq_X g$:
    $\Longrightarrow \exists i \in M_X(f) : \forall j \in M_X(g) : i \not\leq_X j$ $(cp_{\preccurlyeq_X})$
    $\Longrightarrow \exists i \in M(f) : \forall j \in M(g) : i \not\leq j$. (Sum)

  Hence in all cases, $\neg\forall i \in M(f) : \exists j \in M(g) : i \leq j$. ■

D.4 Prop/1

The boolean closure of a given logic. It can also be seen as the usual propositional logic, whose atoms are replaced by the formulas of a logic given as a parameter.

D.4.1 Parameters

$A \in \mathbb{L}$**:** a logic, whose formulas play the role of atoms in propositional logic.

D.4.2 Syntax

AS is the smallest set of formulas that contains AS_A (atoms), 1 (true), 0 (false), and is closed by the application of the unary operator $\neg$ (negation), and the binary operators $\wedge$ (conjunction) and $\vee$ (disjunction). ASK is the subset of propositions whose atoms are all in ASK_A. $TELL$ is the subset of propositions, whose negative atoms are all in ASK_A, whose positive atoms are all in $TELL_A \cup ASK_A$, and such that every conjunctive clause contains a positive literal in $TELL_A$. Positive atoms are those under the scope of an even number of negations; and negative atoms are those under the scope of an odd number of negations.

D.4.3 Semantics

domain: $I = I_A$.

satisfaction:

$$\begin{array}{lll} i \models a & =_{def} & i \models_A a \\ i \models 1 & =_{def} & true \\ i \models 0 & =_{def} & false \\ i \models \neg f & =_{def} & i \not\models f \\ i \models f \wedge g & =_{def} & i \models f \text{ and } i \models g \\ i \models f \vee g & =_{def} & i \models f \text{ or } i \models g \end{array}$$

D.4.4 Operations

subsumption: $f \sqsubseteq g$ is true iff there exists a proof of the sequent $\vdash \neg f \vee g$ in the sequent calculus of Table D.1.

In the rules, Δ is always a set of literals (i.e., atomic formulas or negations of atomic formulas), Γ is a sequence of propositions, L is a literal, X is a proposition, β is the disjunction of $\beta_1 \ldots \beta_n$, α is the conjunction of $\alpha_1 \ldots \alpha_n$, and $\overline{L}$ denotes the negation of L ($\overline{a} := \neg a$ and $\overline{\neg a} := a$). The tautology 1 is equivalent to the empty conjunction, and the contradiction 0 is equivalent to the empty disjunction.

conjunction: $f \sqcap g =_{def} f \wedge g$.

disjunction: $f \sqcup g =_{def} f \vee g$.

$\top$-Axiom:	$\neg b, \Delta \vdash \Gamma$	if $def_{\top_A}$ and $\top_A \sqsubseteq_A b$
$\bot$-Axiom:	$a, \Delta \vdash \Gamma$	if $def_{\bot_A}$ and $a \sqsubseteq_A \bot_A$
$\sqsubseteq$ -Axiom:	$a, \neg b, \Delta \vdash \Gamma$	if $a \sqsubseteq_A b$
$\sqcap$-Rule:	$\frac{a \sqcap_A b, \Delta \vdash \Gamma}{a, b, \Delta \vdash \Gamma}$	if $def_{\sqcap}(a, b)$
$\sqcup$-Rule:	$\frac{\neg(a \sqcup_A b), \Delta \vdash \Gamma}{\neg a, \neg b, \Delta \vdash \Gamma}$	
$\neg\neg$-Rule:	$\frac{\Delta \vdash X, \Gamma}{\Delta \vdash \neg\neg X, \Gamma}$	literal-Rule: $\frac{\overline{L}, \Delta \vdash \Gamma}{\Delta \vdash L, \Gamma}$
β-Rule:	$\frac{\Delta \vdash \beta_1, \ldots, \beta_n, \Gamma}{\Delta \vdash \beta, \Gamma}$	α-Rule: $\frac{\Delta \vdash \alpha_1, \Gamma \ \ldots \ \Delta \vdash \alpha_n, \Gamma}{\Delta \vdash \alpha, \Gamma}$

Table D.1: Sequent calculus for deduction in propositional logic.

tautology: $\top =_{def} 1$.

contradiction: $\bot =_{def} 0$.

D.4.5 Properties

Definition 9 *A sequent $\Delta \vdash \Gamma$ is valid iff $\bigcap_{\delta \in \Delta} M(\delta) \subseteq \bigcup_{\gamma \in \Gamma} M(\gamma)$.*

Definition 10 *The expression $Trees(\Delta \vdash \Gamma)$ denotes the set of all fully developed trees whose root is the sequent $\Delta \vdash \Gamma$.*

Definition 11 *A fully developed tree t is said* open*, denoted by $open(t)$, iff it contains an open sequent (necessary a leaf from definition of an open sequent), i.e. $\exists A, \overline{B} \vdash\in t : open(A, B)$.*

- $df \Leftarrow df_A$
- $cs_{\sqsubseteq} \Leftarrow cs_{\sqsubseteq_A} \wedge cp_{\sqcap_A} \wedge cs_{\sqcup_A} \wedge cp_{\top_A} \wedge cs_{\bot_A}$

  Proof:

  – Let us show that $\sqsubseteq$-Axiom is valid:
  $a \sqsubseteq_A b \Longrightarrow M_A(a) \subseteq M_A(b)$ $cs_{\sqsubseteq_A}$
  $\Longrightarrow M(a) \subseteq M(b) \Longrightarrow \forall i \in I : i \models a \Rightarrow i \models b$

$\Longrightarrow \forall i \in I : i \not\models a \vee i \not\models \neg b$ Semantics of negation
$\Longrightarrow$ the sequent $a, \neg b, \Delta \vdash \Gamma$ is valid.

The $\top$-Axiom and $\bot$-Axiom are valid as a corollary: replace a by $\top_A$ for $\top$-Axiom, and b by $\bot_A$ for $\bot$-Axiom, along with $cp_{\top_A}$ and $cs_{\bot_A}$.

– Let us show that the $\sqcap$-Rule preserves validity. Assume that $a \sqcap_A b$ is defined and the sequent $a \sqcap_A b, \Delta \vdash \Gamma$ is valid, then for all $i \in I$

  * either $\exists X \in \Gamma : i \models X \Longrightarrow$ the sequent $a, b, \Delta \vdash \Gamma$ is valid
  * or $\exists L \in \Delta : i \not\models L \Longrightarrow$ the sequent $a, b, \Delta \vdash \Gamma$ is valid
  * or $i \not\models a \sqcap_A b \Longrightarrow i \notin M(a \sqcap_A b) \Longrightarrow i \notin M_A(a \sqcap_A b)$
    $\Longrightarrow i \notin M_A(a) \cap M_A(b)$ $(cp_{\sqcap_A})$
    $\Longrightarrow i \notin M_A(a) \Longrightarrow i \notin M(a) \Longrightarrow i \not\models a$
    $\Longrightarrow$ the sequent $a, b, \Delta \vdash \Gamma$ is valid.

– Let us show that the $\sqcup$-Rule preserves validity. Assume the sequent $\neg(a \sqcup_A b), \Delta \vdash \Gamma$ is valid, then for all $i \in I$

  * either $\exists X \in \Gamma : i \models X \Longrightarrow$ the sequent $\neg a, \neg b, \Delta \vdash \Gamma$ is valid
  * or $\exists L \in \Delta : i \not\models L \Longrightarrow$ the sequent $\neg a, \neg b, \Delta \vdash \Gamma$ is valid
  * or $\exists c \in a \sqcup_A b : i \not\models \neg c$
    $\Longrightarrow \exists c \in a \sqcup_A b : i \models c$
    $\Longrightarrow \exists c \in a \sqcup_A b : i \in M(c)$
    $\Longrightarrow \exists c \in a \sqcup_A b : i \in M_A(c)$
    $\Longrightarrow i \in \bigcup_{c \in a \sqcup_A b} M_A(c)$
    $\Longrightarrow i \in M_A(a) \cup M_A(b)$ $(cs_{\sqcup_A})$
    $\Longrightarrow i \not\models \neg a \vee i \not\models \neg b$
    $\Longrightarrow \neg a, \neg b, \Delta \vdash \Gamma$ is valid.

– It is easy to recognize that the inference rules $\neg\neg$-Rule, β-Rule, α-Rule and literal-Rule also preserve validity. As a consequence, every provable sequent is valid.

Now for any $f, g \in AS$, if $f \sqsubseteq g$ then $\vdash \neg f \vee g$ is a provable sequent
$\Longrightarrow$ this sequent is valid
$\Longrightarrow \bigcap_{\delta \in \emptyset} M(\delta) \subseteq \bigcup_{\gamma \in \{\neg f \vee g\}} M(\gamma) \Longrightarrow M(\neg f \vee g) = I$
$\Longrightarrow (I \setminus M(f)) \cup M(g) = I \Longrightarrow M(f) \subseteq M(g)$.
This proves $cs_{\sqsubseteq}$.

■

- $cp_{\sqsubseteq}(f, g) \Leftarrow cs_{\sqcap_A} \wedge cp_{\sqcup_A} \wedge$
  $\forall t \in \mathit{Trees}(\vdash \neg f \vee g) :$
  $\mathit{open}(t) \Rightarrow \exists A, \overline{B} \vdash\in t : \mathit{open}_A(A, B) \wedge \mathit{reduced}_A(A, B)$

Proof:

– We first prove that the backward-chaining interpretation of the above inference system terminates for all root sequent. For this proof we need a total ordering of sequents. Every formula being either a literal L, a double negation $\neg\neg X$, a conjunction α or a disjunction β, one defines an integral measure m for every formula in $AS_{prop(A)}$ as follows: $m(\alpha) = 1 + m(\alpha_1) + m(\alpha_2)$, $m(\neg\neg X) = 1 + m(X)$, $m(\beta) = 1 + m(\beta_1) + m(\beta_2)$, and $m(L) = 1$.

  This measure is extended to sequences of propositions Γ, to sequences of literals Δ, and to full sequents $\Delta \vdash \Gamma$ as follows: $m(\Gamma) = \sum_{X\in\Gamma} m(X)$, $m(\Delta) = \sum_{L\in\Delta} m(L)$, $m(\Delta \vdash \Gamma) = (m(\Gamma), m(\Delta))$.

  Finally, sequents are totally ordered according to a lexicographic ordering $<$ on $\mathbb{N}^2$. We observe that for every deduction rule $\frac{Seq_1}{Seq_2}$, $m(Seq_1) < m(Seq_2)$ holds. So, every proof tree is finite. In other words, the backward-chaining procedure always terminates.

– Now, one proves that $\sqcap$-Rule preserves non-validity. Let us assume that $a \sqcap_A b, \Delta \vdash \Gamma$ is not valid.
  Then $\exists i \in I : (i \models a \sqcap_A b$ and $\forall L \in \Delta : i \models L$ and $\forall X \in \Gamma : i \not\models X)$
  $\Longrightarrow \exists i \in I : (i \models a$ and $i \models b$ and $\forall L \in \Delta : i \models L$ and $\forall X \in \Gamma : i \not\models X$ $\quad cs_{\sqcap_A}$
  $\Longrightarrow$ the sequent $a, b, \Delta \vdash \Gamma$ is not valid.

– Now, one proves that $\sqcup$-Rule preserves non-validity. Let us assume that $\neg(a \sqcup_A b), \Delta \vdash \Gamma$ is not valid.
  Then $\exists i \in I : (\forall c \in a \sqcup_A b : i \models \neg c, \forall L \in \Delta : i \models L, \forall X \in \Gamma : i \not\models X)$
  $\Longrightarrow \exists i \in I : \forall c \in a \sqcup_A b : i \not\models c$
  $\Longrightarrow \exists i \in I : \nexists c \in a \sqcup_A b : i \models c$
  $\Longrightarrow \exists i \in I : i \notin \bigcup_{c\in a\sqcup_A b} M(c)$
  $\Longrightarrow \exists i \in I : i \notin M(a) \cup M(b)$ $\quad (cp_{\sqcup_A})$
  $\Longrightarrow \exists i \in I : i \models \neg a, i \models \neg b$
  $\Longrightarrow \neg a, \neg b, \Delta \vdash \Gamma$ is not valid.

– It is easy to check that $\neg\neg$-Rule, α-Rule, β-Rule and literal-Rule also preserve non-validity.

– If $f \sqsubseteq g$ then $\exists t \in Trees(\vdash \neg f \vee g) : \neg open(t)$, i.e. t is a proof of $\vdash \neg f \vee g$
  $\Longrightarrow f \sqcup g$ Definition of $\sqsubseteq$
  $\Longrightarrow M(f) \subseteq M(g)$ $\quad cs_{\sqsubseteq}$
  $\Longrightarrow M(f) \subseteq M(g) \Rightarrow f \sqsubseteq g \Longrightarrow cp_{\sqsubseteq}(f, g)$.
  Else $f \not\sqsubseteq g$, which implies $\forall t \in Trees(\vdash \neg f \vee g) : open(t)$
  $\Longrightarrow \exists A, \overline{B} \vdash\in t : open_A(A, B) \wedge reduced_A(A, B)$ from hypotheses
  $\Longrightarrow \exists A, \overline{B} \vdash\in t : \bigcap_{a\in A} M_A(a) \not\subseteq \bigcup_{b\in B} M_A(b)$ Definition of *reduced*
  $\Longrightarrow \exists A, \overline{B} \vdash\in t : \bigcap_{a\in A} M(a) \cap \bigcap_{b\in B} M(\neg b) \not\subseteq \bigcup_{x\in\emptyset} M(x)$
  $\Longrightarrow$ there is a non-valid sequent $A, \overline{B} \vdash$ in t Definition 9
  $\Longrightarrow$ the sequent $\vdash \neg f \vee g$ is non-valid because all rules preserves the non-validity (see above)
  $\Longrightarrow M(f) \not\subseteq M(g)$.

Hence $cp_{\sqsubseteq}$ (the absence of proof entails the non-validity). ■

- $cp_{\sqsubseteq} \Leftarrow cs_{\sqcap_A} \wedge cp_{\sqcup_A} \wedge reduced_A$

  Proof: To be done (should be easy from previous and next proofs).

- $cp'_{\sqsubseteq} \Leftarrow cs_{\sqcap_A} \wedge cp_{\sqcup_A} \wedge reduced'_A$

  Proof: Let $f \in TELL, g \in ASK$ and $t \in Trees(\neg f \vee g)$.

  From the inference rules, it follows that forall $A, \overline{B} \vdash\in t$, A contains only positive atoms of f and negative atoms of g. Reciprocally, B contains only negative atoms of A and positive atoms of g. Hence B is included in ASK_A, and A is included in $TELL_A \cup ASK_A$. Moreover, from the definitions of syntax and inference rules, we also have that A always contains an atom in $TELL_A$. In summary
  $\forall A, \overline{B} \vdash\in t : (\exists a \in A : a = \bot_A \vee a \in TELL_A) \wedge (\forall a \in A : a \in TELL_A \cup ASK_A \cup \{\bot_A\}) \wedge (\forall b \in B : b \in ASK_A)$
  $\Longrightarrow \forall A, \overline{B} \vdash\in t : reduced_A(A, B)$ $(reduced'_A)$
  $\Longrightarrow open(t) \Rightarrow \exists A, \overline{B} \vdash\in t : open_A(A, B) \wedge reduced_A(A, B)$
  $\Longrightarrow \forall t \in Trees(\vdash \neg f \vee g) : open(t) \Rightarrow \exists A, \overline{B} \vdash\in t : open_A(A, B) \wedge reduced_A(A, B)$
  $\Longrightarrow cp(f, g).$ $(cs_{\sqcap_A}, cp_{\sqcup_A})$ ■

- $def_{\sqcap} \wedge cs_{\sqcap} \wedge cp_{\sqcap}$
- $def_{\sqcup} \wedge cs_{\sqcup} \wedge cp_{\sqcup}$
- $def_{\top} \wedge cp_{\top}$
- $def_{\bot} \wedge cs_{\bot}$
- $reduced(F, G) \Leftarrow \forall f \in F, g \in G : cp_{\sqsubseteq}(f, g)$
  $reduced \Leftarrow cp_{\sqsubseteq}$
  $reduced' \Leftarrow cp'_{\sqsubseteq}$

  Proof:

  Assume we have $F \not\sqsubseteq^* G$, $closed_{\sqcap}(F)$, $closed_{\sqcup}(G)$.
  Then neither conjunction on F, nor disjunction on G can be applied. As these operations are totally defined in *Prop*, we have $F = \{f\}$, and $G = \{g\}$.
  Then we also have $f \not\sqsubseteq g$ Definition of $\sqsubseteq^*$
  $\Longrightarrow M(f) \not\subseteq M(g)$ if $cp_{\sqsubseteq}(f, g)$
  $\Longrightarrow \bigcap_{f \in F} M(f) \not\subseteq \bigcup_{g \in G} M(g)$. ■

Appendix E

Adaptors

The Adaptors help to make a logic satisfy some properties. They are usually inserted lately in the process of designing a logic, after the syntax and semantics structures have been stabilized. So, they do not usually change a lot the syntax, but may change the semantics in a significant way (while preserving the existing structure).

E.1 Set/1

This functor extends the interpretation domain of a logic to its powerset. This has the side effect of replacing the need for the property *reduced* by the weaker property *reduced.right*. A common use example is between the functor *Prop* and a concrete domain or a logic of values.

E.1.1 Parameters

$E \in \mathbb{L}$**:** a logic of elements.

E.1.2 Syntax

$AS = AS_E$
$TELL =_{def} TELL_E$
$ASK =_{def} ASK_E$

E.1.3 Semantics

domain: $I = 2^{I_E} \setminus \{\emptyset\}$.

satisfaction: $i \models f =_{def} \exists i_E \in i : i \models_E f$.

E.1.4 Operations

subsumption: $f \sqsubseteq g =_{def} f \sqsubseteq_E g$,

tautology: $\top =_{def} \top_E$,

contradiction: $\bot =_{def} \bot_E$,

disjunction: $f \sqcup g =_{def} f \sqcup_E g$.

E.1.5 Properties

- $df \Leftarrow df_E$
- $st(f) \Leftarrow st_E(f)$
  $st \Leftarrow st_E$
  $st' \Leftarrow st'_E$
  Proof: $st_E(f) \Longrightarrow M_E(f) \neq \emptyset$
  $\Longrightarrow \exists i_E \in I_E : i_E \models_E f$
  $\Longrightarrow \exists i = \{i_E\} \in I : i \models f$
  $\Longrightarrow M(f) \neq \emptyset \Longrightarrow st(f)$. ■

- $cs_{\sqsubseteq} \Leftarrow cs_{\sqsubseteq_E}$
  Proof: $f \sqsubseteq g \Longrightarrow f \sqsubseteq_E g$
  $\Longrightarrow M_E(f) \subseteq M_E(g)$ $(cs_{\sqsubseteq_E})$
  $\Longrightarrow \forall i_E : i_E \models_E f \Rightarrow i_E \models_E g$
  $\Longrightarrow \forall i \in I : (\exists i_E \in i : i_E \models_E f) \Rightarrow (\exists i_E \in i : i_E \models_E g)$
  $\Longrightarrow \forall i \in I : i \models f \Rightarrow i \models g$
  $\Longrightarrow M(f) \subseteq M(g)$. ■

- $cp_{\sqsubseteq} \Leftarrow cp_{\sqsubseteq_E}$
  Proof: $M(f) \subseteq M(g) \Longrightarrow \forall i \in I : i \models f \Rightarrow i \models g$
  $\Longrightarrow \forall i \in I : (\exists i_E \in i : i_E \models_E f) \Rightarrow (\exists i_E \in I : i_E \models_E g)$
  $\Longrightarrow \forall \{i_E\} \in I : i_E \models_E f \Rightarrow i_E \models_E g$
  $\Longrightarrow M_E(f) \subseteq M_E(g) \Longrightarrow f \sqsubseteq_E g$ $(cp_{\sqsubseteq_E})$
  $\Longrightarrow f \sqsubseteq g$. ■

- $cp_{\top} \Leftarrow cp_{\top_E}$
  Proof: $M_E(\top_E) = I_E$ $(cp_{\top_E})$
  $\Longrightarrow \forall i_E \in I_E : i_E \models_E \top_E$
  $\Longrightarrow \forall i \in I : \exists i_E \in i : i_E \models_E \top_E$
  $\Longrightarrow \forall i \in I : i \models \top_E$
  $\Longrightarrow M(\top) = M(\top_E) = I$. ■

- $cs_{\bot} \Leftarrow cs_{\bot_E}$
  Proof: $M_E(\bot_E) = \emptyset$ $(cp_{\bot_E})$
  $\Longrightarrow \forall i_E \in I_E : i_E \not\models_E \bot_E$
  $\Longrightarrow \forall i \in I : \forall i_E \in i : i_E \not\models_E \bot_E$
  $\Longrightarrow \forall i \in I : i \not\models \top_E$
  $\Longrightarrow M(\bot) = M(\bot_E) = \emptyset$. ■

- $cs_{\sqcap}$

- $cp_{\sqcap}$

- $cs_{\sqcup} \Leftarrow cs_{\sqcup_E}$
  Proof: Let $f, g \in AS$
  $\Longrightarrow f, g \in AS_E$
  $\Longrightarrow \bigcup_{h \in f \sqcup_E g} M_E(h) \subseteq M_E(f) \cup M_E(g)$ $(cs_{\sqcup_E})$
  $\Longrightarrow \forall i_E \in I_E : (\exists h \in f \sqcup_E g : i_E \models_E h) \Rightarrow i_E \models_E f \vee i_E \models_E g$
  $\Longrightarrow \forall i \in I : (\exists i_E \in i, h \in f \sqcup_E g : i_E \models_E h) \Rightarrow (\exists i_E \in i : i_E \models_E f \vee i_E \models_E g)$
  $\Longrightarrow \forall i \in I : (\exists h \in f \sqcup_E g, i_E \in i : i_E \models_E h) \Rightarrow (\exists i_E \in i : i_E \models_E f) \vee (\exists i_e \in i : i_E \models_E g)$

$\Longrightarrow \forall i \in I : (\exists h \in f \sqcup g : i \models h) \Rightarrow i \models f \vee i \models g$
$\Longrightarrow \bigcup_{h \in f \sqcup g} M(h) \subseteq M(f) \cup M(g)$. ■

- $cp_{\sqcup} \Leftarrow cp_{\sqcup_E}$
  Proof: The proof can be derived from the previous proof for $cs_{\sqcup}$ by replacing $\subseteq$ by $\supseteq$, and $\Rightarrow$ by $\Leftarrow$. ■

- $reduced(F, G) \Leftarrow \forall f \in F : reduced_E(\{f\}, G)$
  $reduced \Leftarrow reduced.right$
  $reduced' \Leftarrow reduced.right$

  Proof: $\bigcap_{f \in F} M(f) \subseteq \bigcup_{g \in G} M(g) \wedge closed_{\sqcup}(G)$
  $\Longrightarrow closed_{\sqcup_E}(G)$
  and $\forall i \in I : (\forall f \in F : i \models f) \Rightarrow (\exists g \in G : i \models g)$
  $\Longrightarrow \forall i \in I : (\forall f \in F : \exists i_E \in i : i_E \models_E f) \Rightarrow (\exists g \in G : \exists i_E \in i : i_E \models_E g)$
  $\Longrightarrow \forall i \in I : (\forall f \in F : \exists i_E \in i : i_E \in M_E(f)) \Rightarrow (\exists i_E \in i : i_E \in \bigcup_{g \in G} M_E(g))$.

  Now suppose that $\forall f \in F : M_E(f) \not\subseteq \bigcup_{g \in G} M_E(g)$
  $\Longrightarrow \forall f \in F : \exists i_E : i_E \models_E f \wedge i_E \notin \bigcup_{g \in G} M_E(g)$
  This allows to build an $i \in I$ s.t. $\forall f \in F : \exists i_E \in i : i_E \models_E f \wedge i_E \notin \bigcup_{g \in G} M_E(g)$, i.e. i contains an i_E for each $f \in F$, and only that.
  $\Longrightarrow (\forall f \in F : \exists i_E \in i : i_E \in M_E(f)) \wedge \neg(\exists i_E \in i : i_E \in \bigcup_{g \in G} M_E(g)$, which contradicts the above implication for all $i \in I$.

  Hence $\exists f \in F : M_E(f) \subseteq \bigcup_{g \in G} M_E(g)$
  $\Longrightarrow \exists f \in F : \{f\} \sqsubseteq^*_E G)$ $(reduced_E(\{f\}, G))$
  So we have one of the following:

  - $def(\bot_E) \wedge f \sqsubseteq_E \bot_E \Longrightarrow def(\bot) \wedge f \sqsubseteq \bot \Longrightarrow F \sqsubseteq^* G)$
  - $\exists g \in G : def(\top_E) \wedge \top_E \sqsubseteq_E g \Longrightarrow \exists g \in G : def(\top) \wedge \top \sqsubseteq g \Longrightarrow F \sqsubseteq^* G$
  - $\exists g \in G : f \sqsubseteq_E g \Longrightarrow \exists g \in G : f \sqsubseteq g \Longrightarrow F \sqsubseteq^* G$

  In conclusion we have $F \sqsubseteq^* G$ in all cases, which terminates this proof. ■

E.2 Bottom/1

This functor adds a bottom to the language, and make it the result of the conjunction of a *TELL*-formula and a non-subsuming *ASK*-formula. This functor is useful to ensure the property *reduced'*, given the properties *sg'*, and $cp'_{\sqsubseteq}$.

E.2.1 Parameters

$X \in \mathbb{L}$**:** a logic, whose *TELL*-formulas have a single model.

E.2.2 Syntax

$AS =_{def} AS_X \uplus \{0\}$.
$TELL =_{def} TELL_X$.
$ASK =_{def} ASK_X$.

E.2.3 Semantics

domain: $I =_{def} I_X$.

satisfaction: $i \models f =_{def} i \models_X f$, for all $f \in AS_X$.

E.2.4 Operations

subsumption:

$$f \sqsubseteq g =_{def} \begin{cases} \mathit{true} & \text{if } f = 0 \\ f \sqsubseteq_X g & \text{if } f, g \in AS_X \\ \mathit{false} & \text{otherwise} \end{cases}$$

tautology: $\top =_{def} \top_X$.

contradiction: $\bot =_{def} 0$.

conjunction:

$$f \sqcap g =_{def} \begin{cases} 0 & \text{if } f = 0 \\ 0 & \text{if } g = 0 \\ 0 & \text{if } f \in TELL \wedge g \in ASK \wedge f \not\sqsubseteq g \\ 0 & \text{if } g \in TELL \wedge f \in ASK \wedge g \not\sqsubseteq f \\ f \sqcap_X g & \text{if } f, g \in AS_X \end{cases}$$

E.2.5 Properties

- $df \Leftarrow {df}_X$
- ${st}' \Leftarrow {st'}_X$

  Proof: $d \in TELL \Longrightarrow d \in TELL_X \Longrightarrow M_X(d) \neq \emptyset$ $({st'}_X)$
  $\Longrightarrow M(d) \neq \emptyset$. ■

- ${sg}' \Leftarrow {sg'}_X$

  Proof: $d \in TELL \Longrightarrow d \in TELL_X \Longrightarrow Card(M_X(d))$ $({sg'}_X)$
  $\Longrightarrow Card(M(d)) = 1$. ■

- ${cs}_{\sqsubseteq} \Leftarrow {cs}_{\sqsubseteq_X}$

  Proof: Let $f, g \in TELL$, such that $f \sqsubseteq g$. Then

  - either $f = 0$: $M(f) = \emptyset \Longrightarrow M(f) \subseteq M(g)$,
  - or $f, g \in AS_X$, and $f \sqsubseteq_X g$: $M_X(f) \subseteq M_X(g)$ $({cs}_{\sqsubseteq_X})$
    $\Longrightarrow M(f) \subseteq M(g)$. ■

- ${cp'}_{\sqsubseteq} \Leftarrow {cp'}_{\sqsubseteq_X}$

  Proof: Let $d \in TELL, x \in ASK$, such that $M(d) \supseteq M(x)$.
  Then $d \in TELL_X, x \in ASK_X$, $M_X(d) \subseteq M_X(x)$
  $\Longrightarrow d \sqsubseteq_X x$ $({cp'}_{\sqsubseteq_X})$
  $\Longrightarrow d \sqsubseteq x$. ■

- ${cp}_{\top} \Leftarrow {cp}_{{\top}_X}$

  Proof: $M(\top) = M(\top_X) = M_X(\top_X) = I_X$ $({cp}_{{\top}_X}) = I$. ■

- ${cs}_{\bot}$

  Proof: $M(\bot) = M(0) = \emptyset$. ■

- ${defst}_{\sqcap} \Leftarrow {defst}_{{\sqcap}_X}$

  Proof: Let $f, g \in AS$, s.t. $M(f) \cap M(g) \neq \emptyset$. Then $f \neq 0$, and $g \neq 0$, because $M(0) = \emptyset$. So, $f, g \in AS_X$, and $M_X(f) \cap M_X(g) \neq \emptyset$
  $\Longrightarrow def(f \sqcap_X g)$ $({defst}_{{\sqcap}_X})$
  $\Longrightarrow def(f \sqcap g)$. ■

- $cs_\sqcap \Leftarrow cs_{\sqcap_X}$

  Proof: Let $f, g \in AS$ s.t. $f \sqcap g$ is defined:

  - $f = 0$: $M(0) = \emptyset$,
  - $g = 0$: idem,
  - $f \in TELL, g \in ASK, f \not\sqsubseteq g$: idem,
  - $f \in ASK, g \in TELL, g \not\sqsubseteq f$: idem,
  - $f, g \in AS_X$: $M_X(f \sqcap_X g) \subseteq M_X(f) \cap M_X(g)$ $(cs_{\sqcap_X})$
    $\Longrightarrow M(f \sqcap_X g) \subseteq M(f) \cap M(g)$. ■

- $cp_\sqcap \Leftarrow cp_{\sqcap_X} \wedge {sg'}_X \wedge {cp'}_{\sqsubseteq_X}$

  Proof: Let $f, g \in AS$ s.t. $f \sqcap g$ is defined:

  - $f = 0$: $M(f) = M(0) = \emptyset \Longrightarrow M(f) \cap M(g) = \emptyset$,
  - $g = 0$: idem,
  - $f \in TELL, g \in ASK, f \not\sqsubseteq g$: $f \in TELL_X, g \in ASK_X, f \not\sqsubseteq_X g$
    $\Longrightarrow M_X(f) \not\subseteq M_X(g)$ $({cp'}_{\sqsubseteq_X})$
    $\Longrightarrow M_X(f) \cap M_X(g) = \emptyset$ $({sg'}_X)$
    $\Longrightarrow M(f) \cap M(g) = \emptyset$,
  - $f \in ASK, g \in TELL, g \not\sqsubseteq f$: idem,
  - $f, g \in AS_X$: $M_X(f \sqcap_X g) \supseteq M_X(f) \cap M_X(g)$ $(cp_{\sqcap_X})$
    $\Longrightarrow M(f \sqcap_X g) \supseteq M(f) \cap M(g)$. ■

- $reduced' \Leftarrow {df}_X \wedge {sg'}_X \wedge {cs}_{\sqsubseteq_X} \wedge {cp'}_{\sqsubseteq_X}$

  Proof: Let $F, G \subseteq AS$, s.t. $\bigcap_{f \in F} M(f) \subseteq \bigcup_{g \in G} M(g)$, $F \subseteq TELL \cup ASK \cup \{\bot\}$, $F \cap (TELL \cup \{\bot\}) \neq \emptyset$, $closed_\sqcap(F)$, and $G \subseteq ASK$.

  - either $\bot = 0 \in F$: $\exists f \in F : f \sqsubseteq \bot$ (because $0 \sqsubseteq 0$)
    $\Longrightarrow F \sqsubseteq^* G$,
  - or $F \subseteq TELL \cup ASK = TELL_X \cup ASK_X = ASK_X$, $\exists d \in F : d \in TELL$: $({df}_X)$
    $\Longrightarrow \forall f \in F : d \sqsubseteq_X f \vee d \not\sqsubseteq_X f$
    $\Longrightarrow \forall f \in F : d \sqsubseteq f \vee (d \in TELL_X \wedge f \in ASK_X \wedge d \not\sqsubseteq_X f)$
    $\Longrightarrow \forall f \in F : d \sqsubseteq f \vee def(d \sqcap f)$
    $\Longrightarrow \forall f \in F : d \sqsubseteq f$ $(closed_\sqcap(F))$
    $\Longrightarrow \forall f \in F : M(d) \subseteq M(f)$ $(cs_{\sqsubseteq_X})$
    $\Longrightarrow \bigcap_{f \in F} M(f) = M(d) = M_X(d) = \{i_d\}$. $({sg'}_X)$
    Now, $\bigcap_{f \in F} M(f) \subseteq \bigcup_{g \in G} M(g)$
    $\Longrightarrow i_d \in \bigcup_{g \in G} M(g)$

$$
\begin{aligned}
&\Longrightarrow \exists g \in G : g \in ASK_X, i_d \in M_X(g) \\
&\Longrightarrow \exists d \in F, g \in G : d \in TELL_X, g \in ASK_X, M_X(d) \subseteq M_X(g) \\
&\Longrightarrow \exists d \in F, g \in G : d \sqsubseteq_X g && (cp'_{\sqsubseteq_X}) \\
&\Longrightarrow \exists d \in F, g \in G : d \sqsubseteq g \\
&\Longrightarrow F \sqsubseteq^* G. && \blacksquare
\end{aligned}
$$

E.3 Single/1

This functor ensures that the property sg' is satisfied. The property df is required on the argument logic in order to satisfy the partial completeness $cp'_{\sqsubseteq}$. This functor is based on the epistemic closure [Fer06], which is itself derived from the logic AIK (*All I Know* [Lev90]).

E.3.1 Parameters

$X \in \mathbb{L}$**:** a logic.

E.3.2 Syntax

$$\begin{array}{lcl} AS & \rightarrow & AS_X \quad | \quad [AS_X] \\ TELL & \rightarrow & [TELL_X] \\ ASK & \rightarrow & ASK_X \quad | \quad [TELL_X] \end{array}$$

The square brackets denote the modal operator O that is specific to AIK. The usual modal operator K implicitly applies in other cases.

E.3.3 Semantics

domain: $I =_{def} 2^{I_X}$.

satisfaction:

$$\begin{array}{ll} i \models f =_{def} i \subseteq M_X(f), & M(f) =_{def} 2^{M_X(f)} \\ i \models [f] =_{def} i = M_X(f), & M([f]) =_{def} \{M_X(f)\} \end{array}$$

E.3.4 Operations

subsumption:

$\sqsubseteq$	g	$[g]$
f	$f \sqsubseteq_X g$	false
$[f]$	$f \sqsubseteq_X g$	$f \equiv_X g$

tautology: $\top =_{def} \top_X$.

E.3.5 Properties

- df
- st'

  Proof: Trivial from the definition of semantics. ■

- sg'

  Proof: Trivial from the definition of semantics. ■

- $cs_{\sqsubseteq} \Leftarrow cs_{\sqsubseteq_X}$

  Proof: Let $f, g \in AS$, s.t. $f \sqsubseteq g$. Then

  - either $f, g \in AS_X$: $f \sqsubseteq_X g$
    $\Longrightarrow M_X(f) \subseteq M_X(g)$ $(cs_{\sqsubseteq_X})$
    $\Longrightarrow 2^{M_X(f)} \subseteq 2^{M_X(g)} \Longrightarrow M(f) \subseteq M(g)$,
  - or $f = [f'] \in [AS_X], g \in AS_X$: $f' \sqsubseteq_X g$
    $\Longrightarrow M_X(f') \subseteq M_X(g)$ $(cs_{\sqsubseteq_X})$
    $\Longrightarrow \{M_X(f')\} \subseteq 2^{M_X(g)} \Longrightarrow M(f) \subseteq M(g)$,
  - or $f = [f'], g = [g'] \in [AS_X]$: $f' \equiv_X g' \Longrightarrow f' \sqsubseteq_X g' \wedge g' \sqsubseteq f'$
    $\Longrightarrow M_X(f') \subseteq M(g') \wedge M_X(g') \subseteq M(f')$ $(cs_{\sqsubseteq_X})$
    $\Longrightarrow M_X(f') = M_X(g') \Longrightarrow M(f) = M(g)$. ■

- ${cp'}_{\sqsubseteq} \Leftarrow df_X \wedge {cp'}_{\sqsubseteq_X}$

  Proof: Let $d = [d'] \in TELL, d' \in TELL_X, x \in ASK$, s.t. $M(d) \subseteq M(f)$. There are 2 cases:

  - either $x \in ASK_X$: $\{M_X(d')\} \subseteq 2^{M_X(x)}$
    $\Longrightarrow M_X(d') \subseteq M_X(x) \Longrightarrow d' \sqsubseteq_X x$ $({cp'}_{\sqsubseteq_X})$
    $\Longrightarrow d \sqsubseteq x$,
  - or $x = [x'] \in [TELL_X], x' \in TELL_X$: $\{M_X(d')\} \subseteq \{M_X(x')\}$
    $\Longrightarrow M_X(d') \subseteq M_X(x') \wedge M_X(x') \subseteq M_X(d')$
    $\Longrightarrow d' \sqsubseteq_X x' \wedge x' \sqsubseteq_X d'$ $(df_X, {cp'}_{\sqsubseteq_X})$
    $\Longrightarrow d' \equiv_X x' \Longrightarrow d \sqsubseteq x$. ■

- $cp_{\top} \Leftarrow cp_{\top_X}$

  Proof: $M(\top) = M(\top_X) = 2^{M_X(\top_X)} = 2^{I_X}$ $(cp_{\top_X}) = I$. ■

E.4 Id/1

This functor ensures that the property *df* is satisfied. This is done by adding an identifier in descriptors, and in interpretations. As every identifier maps to only one *TELL*-sub-formula, it becomes possible to have a complete entailment between *TELL*-formulas, and so to include them in the set of *ASK*-formulas.

E.4.1 Parameters

$X \in \mathbb{L}$**:** a logic.

E.4.2 Syntax

$AS =_{def} AS_X \cup (\mathbb{N} \times AS_X)$
$TELL =_{def} \mathbb{N} \times TELL_X$
$ASK =_{def} ASK_X \cup (\mathbb{N} \times TELL_X)$

In addition to these definitions, we state the following invariant:

$$\forall (n, f), (m, g) \in AS : n = m \Rightarrow f = g.$$

E.4.3 Semantics

domain: $I =_{def} \mathbb{N} \times I_X$.

satisfaction:

$$\begin{array}{llll} (m,i) \models f & =_{def} i \models_X f & , & M(f) =_{def} \mathbb{N} \times M_X(f) \\ (m,i) \models (n,f) & =_{def} m = n \wedge i \models_X f, & & M((n,f)) =_{def} \{n\} \times M_X(f) \end{array}$$

E.4.4 Operations

subsumption:

$\sqsubseteq$	g	(m,g)
f	$f \sqsubseteq_X g$	false
(n,f)	$f \sqsubseteq_X g$	$n = m$

tautology: $\top =_{def} \top_X$.

contradiction: $\bot =_{def} \bot_X$.

E.4.5 Properties

- df
- $st \Leftarrow st_X$
- $st' \Leftarrow {st'}_X$
- $sg' \Leftarrow {sg'}_X$

  Proof: Let $(n, f) \in TELL$.
  $M((n, d)) = \{n\} \times M_X(d)$
  $\Longrightarrow Card(M((n, d)) = Card(M_X(d)) = 1.\ (d \in TELL_X, {sg'}_X)$ ■

- $cs_\sqsubseteq \Leftarrow {cs_\sqsubseteq}_X$

  Proof: Let $f, g \in AS$, s.t. $f \sqsubseteq g$. Then

  - either $f, g \in AS_X$: $f \sqsubseteq_X g$
    $\Longrightarrow M_X(f) \subseteq M_X(g)$ $(cs_{\sqsubseteq_X})$
    $\Longrightarrow \mathbb{N} \times M_X(f) \subseteq \mathbb{N} \times M_X(g) \Longrightarrow M(f) \subseteq M(g)$,
  - or $f = (n, f') \in \mathbb{N} \times AS_X, g \in AS_X$: $f' \sqsubseteq_X g$
    $\Longrightarrow M_X(f') \subseteq M_X(g)$ $(cs_{\sqsubseteq_X})$
    $\Longrightarrow \{n\} \times M_X(f') \subseteq \mathbb{N} \times M_X(g) \Longrightarrow M(f) \subseteq M(g)$,
  - or $f = (n, f'), g = (m, g') \in \mathbb{N} \times AS_X$: $n = m \Longrightarrow f' = g'$ (invariant)
    $\Longrightarrow M_X(f') = M_X(g') \Longrightarrow \{n\} \times M_X(f') = \{m\} \times M_X(g')$ $(n = m)$
    $\Longrightarrow M(f) = M(g)$. ■

- ${cp'}_\sqsubseteq \Leftarrow {cp'}_{\sqsubseteq_X}$

  Proof: Let $d = (n, d') \in TELL, d' \in TELL_X, x \in ASK$, s.t. $M(d) \subseteq M(x)$. There are 2 cases:

  - either $x \in ASK_X$: $\{n\} \times M_X(d') \subseteq \mathbb{N} \times M_X(x)$
    $\Longrightarrow M_X(d') \subseteq M_X(x) \Longrightarrow d' \sqsubseteq_X x$ $({cp'}_{\sqsubseteq_X})$
    $\Longrightarrow d \sqsubseteq x$,
  - or $x = (m, x') \in \mathbb{N} \times TELL_X, x' \in TELL_X$: $\{n\} \times M_X(d') \subseteq \{m\} \times M_X(x')$
    $\Longrightarrow n = m \wedge M_X(d') \subseteq M_X(x')$
    $\Longrightarrow d \sqsubseteq x$. ■

- $cp_\top \Leftarrow {cp_\top}_X$

  Proof: $M(\top) = M(\top_X) = \mathbb{N} \times M_X(\top_X) = \mathbb{N} \times I_X\ ({cp_\top}_X) = I$. ■

- $cs_{\perp} \Leftarrow cs_{\perp_X}$

  Proof: $M(\perp) = M(\perp_X) = \mathbb{N} \times M_X(\perp_X) = \mathbb{N} \times \emptyset\ (cs_{\perp_X}) = \emptyset$. ■

E.5 Aik/1 $= \lambda X.Prop(Bottom(Single(X)))$

This combinator is commonly applied in logical information systems on the logic of object descriptions and features. This produces a full querying language with boolean connectives matching set operations on sets of answers, i.e., queries are interpreted under the Closed World Assumption.

Contents

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399